AD-A206 040

RADC-TR-88-168, Vol II (of two)
Final Technical Report
August 1988

A DESIGNERS' GUIDE TO RELIABLE DISTRIBUTED SYSTEMS An Example Design

Honeywell

Anand R. Tripathi, Jonathan Silverman, William T. Wood, Elaine N. Frankowski, Pong-Sheng Wang, Shiva Azadegan, Shiv Seth, Rita Wu, Helmut K. Berg

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 4 03 094

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-88-168, Vol II (of two) has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command and Control

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Directorate of Plans & Programs

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

ADA206040

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-88-168, Vol II (of two)
6a. NAME OF PERFORMING ORGANIZATION Honeywell 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Computer Science Center 10701 Lyndale Ave (South) Bloomington MN 55420	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-82-C-0154
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO: 63728F PROJECT NO.: 2530 TASK NO: 01 WORK UNIT ACCESSION NO: 17

11. TITLE (Include Security Classification)
A DESIGNERS' GUIDE TO RELIABLE DISTRIBUTED SYSTEMS An Example Design

12. PERSONAL AUTHOR(S) Anand R. Tripathi, Jonathan Silverman, William T. Wood, Elaine N. Frankowski, Pong-Sheng Wang, Shiva Azadegan, Shiv Seth, Rita Wu, Helmut K. Berg

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Sep 82 TO Aug 84	August 1988	300

16. SUPPLEMENTARY NOTATION Subcontractors: Information Research Associates - Authors: James C. Browne, James Dutton, Vincent Fernandes, Annette Palmer, Raj Kumar Velpuri
The University of Texas at Austin - Authors: Donald I. Good, Michael K. Smith (See Reverse)

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Distributed Systems; Reliable Systems
12	07		Performance Evaluation; Reliability Evaluation
			Recovery Mechanisms; Atomic Action (See Reverse)

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes an effort to develop a system designers guidebook for designing reliable distributed command and control systems. The guidebook contains a synthesis of reliable system design principles and methods to evaluate distributed system designs for performance, reliability and functional correctness. The approach to developing the system designers guidebook in this effort is example driver. We develop a detailed design of a reliable distributed operating system and evaluate its performance.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315) 330-2158 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 — Previous editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

UNCLASSIFIED

Block 16 SUPPLEMENTARY NOTATION (Continued).

Richard M. Cohen, Lawrence Smith, Lawrence Akers, William Bevier, Miren Carranza, Ann Siebert

Block 18 SUBJECT TERMS (Continued).

Fault-Tolerant Systems	Volidation
Verification	Replication
Commit Protocol	Design Methods
Object-Oriented Systems	Formal Specification

UNCLASSIFIED

VOLUME 2

TABLE OF CONTENTS

CHAPTER 1

ZEUS ARCHITECTURE AND FUNCTIONAL DEFINITION

1.1 INTRODUCTION

Zeus is an object oriented distributed operating system designed to study integration of recovery mechanisms into the designs of distributed command and control systems. The primary goal of the Zeus design is to define reliable object management functions for distributed command and control systems and to evaluate the performance and the correctness of the recovery mechanisms for these functions. Therefore, no implemenation of this design currently exists. The user provided functions support definition of object types, creation of objects, and updating of distributed objects using atomic transactions. the goal of this design is to study the performance characteristics of this design using simulation models and to prove the correctness of the recovery mechanisms using formal methods based on Gypsy language, events and state transition based models , and simulation models. To achieve these goals we have refined the Zeus design to a significantly detailed level. To date we have explored this design only from the viewpoint of these goals. This chapter presents an overview and gives the functional definition of Zeus.

1.2 ZEUS OVERVIEW

Zeus is an example of an object oriented distributed system. Conceptually, therefore, Zeus is a collection of type managers each of which ensures the integrity of some abstract data type. Each type manager is distributed on a number of hosts in the system and the individual instances of the abstract data type (or object instances) may be distributed. A description of Zeus is, therefore, a description of the type managers in the system and a definition of the underlying execution environment for the type managers. The copy of a type manager at a given site enforces part of a global integrity mechanism for that type. The goal of this project is to evaluate integrity mechanisms. Thus, the design of Zeus must be a framework onto which different integrity mechanisms can easily fit. This section provides that framework.

Each type manager has an internal structure that is common to type managers. This includes mechanisms for protection and integrity. There are some types which are necessary to support a user_oriented distributed computing environment in the system. These types are specified in some detail in section 1.2.2. They include symbolic names, principals and authentication, programs, messages, processes and transactions, and unique identifiers.

As a distributed operating system Zeus is designed to be implemented on a variety of configurations. Logically Zeus consists of a number of type managers. Each type manager contains one or more objects of its type. The invocation of type functions against these objects is the means by which computation is achieved in the system. Each type manager in Zeus executes on a virtual machine that is defined by some hardware configuration and by a software kernel. The software kernel allocates resources to the type managers. These include the CPU, volatile and non-volatile storage, and access to the interconnection network that supports remote invocations on type managers.

1.3 FUNCTIONAL DEFINITION OF ZEUS

Zeus consists essentially of a collection of Type Managers (TMs); typically, many different type managers coexist on a host node. The core of the operating system consists of a set of type managers that support capabilities for defining new types and object instances in the system, for authenticating of users, for creating naming environment for each user, and for reliably managing processes and transactions. These system-defined type managers reside at every node in the system.

The lowest level of operating system at each node is called the kernel; the kernel virtualizes the resources at the host so that each type manager can be viewed as having its own virtual processor. The kernel supports interprocess communication, primary storage management, processor scheduling, interfaces to secondary storage devices, and UID generation. As shown in Figure 1-1, all type managers at a node execute over the abstract machine interface provided by the kernel. The kernel multiplexes the processor between the type managers; it also handles all interrupts due to storage devices and the communication devices.

1.3.1 Zeus Kernel

The Zeus kernel provides low level services to the type managers of the system. These services include interprocess communication, storage management and unique identifier (UID) generation. The UID generation in turn depends on the failure detection and recovery of hosts in the Zeus system. The kernel assumes that it is executing on a host that belongs to a cluster. The cluster is assumed to consist of a number of hosts connected by a CSMA/CD (Carrier Sense Multiple Access/Collision Detection) network with the added property of reliable broadcast.

Interprocess communication is achieved by the mechanism of remote procedure call (RPC) which consists of four messages interchanged between caller and callee. These are call, call acknowledge, response and response acknowledge. For each call that is made from or to a type manager the status of the call parameters and status must be stored. To do this each type manager has a call handler to perform this function. The synchronous nature

of the RPC is achieved by the type managers who will first issue a call and then on getting the response will inform the caller of it.

The storage functions of the kernel are performed at the object level. Thus calls to the kernel can retrieve, store and delete objects. Further stable storage operations can be executed by the kernel, where stable storage is implemented using the Lampson scheme [LAMP81].

UID generation is a function used by the RPC and by the type managers so that calls and objects can be uniquely identified. This function must continue despite failure and recovery of hosts. To achieve this the hosts participate in a distributed computation to keep track of active hosts and to let new or recovered hosts join in the UID generation function.

Since communication over the network occurs via packets there must be a message packetization and re-assembly function underlying the RPC function. Below this packet level protocol there is a driver which interfaces with the network controller.

1.3.1.1 Kernel Functions

The kernel interface consists of three parts; remote procedure calls, object storage management and unique identifier generation. Each part has a set of procedures that can be invoked from the type managers or by user processes.

1.3.1.1.1 The Remote Procedure Call

The Remote Procedure Call functions provide the call invoker with the facilities to initiate a call, receive the response to a call and to inquire about a call's status. Similarly, the recipient of a call has the facility to receive a call, make a response to a call, and inquire about a response's status. Functions also exist to cancel a call or retain a call. Each call is identified uniquely in the system by a unique identifier. A request for a call or a response will return to the caller any call or response that is waiting for the caller; it is then the caller's responsibility to deliver the call or response to the correct process. The kernel interface procedures are specified below:

1. procedure make_call (type of caller, source of call, destination of call, call contents, call options, call unique identifer, call status)
   The call contents include the operation to be invoked and the parameters of that operation.
2. procedure get_resp (type of caller, call unique identifier, call s response)
   The call unique identifer and response and output by the kernel. Thus the invoker cannot make a request for a particular response.
3. procedure c_status (type of caller, call identifier,

call status)
The call status is returned and will tell the invoker of the current status of the call, i.e., whether it has been delivered, whether the response to the call has been received or whether the call has failed.
4. procedure make_resp (type of caller, source of call, destination of call, call identifier, response options, response status)
The make_resp procedure has very similar parameters to the make_call procedure.
5. procedure get_call (type of caller, call identifier, call contents)
6. procedure v_status (type of caller, call identifier, response status)
7. procedure kill_call (type of caller, call identifier, call status)
8. procedure keep_call (type of caller, call identifier, call status)
The kill_call and keep call procedures are used to update the local tables for a call.

1.3.1.1.2 Object Storage Management

The Object Storage Management functions permit type managers to store, retrieve and delete objects. Objects can be stored on simple or stable devices; thus the interface has two sets of calls.

The procedure definitions for the storage management functions are:

1. procedure get_obj (type of object to be retrieved, identifier of object to be retrieved, object contents, operation status)
This is for simple object retrieval.
2. procedure put_obj (type of object to be stored, identifier of object to be stored, object contents, operation status)
3. procedure del_obj (type of object to be deleted, identifier of object to be deleted, operation status)
4. procedure stabl_get
This has identical parameters to get_obj.
5. procedure stabl_put
This has identical parameters to put_obj.
6. procedure stabl_del
This has identical parameters to del_obj.

1.3.1.1.3 Unique Identifier Generation

This part of the interface permits an invoker to obtain a new unique identifier, construct an extended unique identifier, obtain the host hint of an object and change the host hint of the object.

The functions for UID generation are:

1. Function get_UID returns UID.
2. Function build_xt (host hint, object type UID, object instance UID, object version UID) returns extended UID
3. Function give_host_hint (object extended UID) returns object host hint
4. Function change_hint (object extended UID, new host hint) returns modified object extended UID.

1.3.1.2 Structure of Zeus Kernel

The Zeus kernal consists of five major components which are: Dispatcher, Operation Switch, Network Handler, Storage Handler and Unique Identifier Generation. Section 1.3.1.2.1 briefly explains the kernal_Resource Management which includes Dispatcher, Storage Handler and Kernal initiator. Section 1.3.1.2.2 presents the Operation Switch. Section 1.3.1.2.3 discusses the Unique Identfier Generation and its two components: Small Stepper and Large Stepper and Section 1.3.1.2.4 presents the Network Handler.

1.3.1.2.1 Zeus Kernel_Resource Management

The task dispatcher schedules the different type managers and handles their requests for resources. The storage handler manages both volatile and non-volatile memory. Storage management in the kernel is minimal. Storage is available in fixed sized blocks and the type managers request one or more of these blocks at any time. A type manager is solely responsible for the data he writes to the blocks of storage. The kernel keeps track of the ownership of blocks of storage. The kernel initiator has two functions. The first function is to restart a host when it recovers from a failure. The second is to initiate a task. Both tasks require a certain amount of housekeeping. Host recovery implies the setting up of tables for the dispatcher of the kernel, using the log for the Type-Type Manager to create, delete, or modify the type managers on the host, and obtaining a new incarnation number and the Small Stepper sequence number. After the above actions are successfully completed, the initiator can hand control to the task dispatcher.

1.3.1.2.2 Operation Switch

The processing of remote procedure calls is the major function of the kernel. Each call is an operation invoked against an object that is held by some type manager. The Zeus design stipulates that each object in the system has a unique identifier. This unique identifier consists of a host hint to speed up object location, a unique type identifier and a unique identifier for a type instance. The composite identifier specified above is called the extended UID of the object. The type and instance components are unique to Zeus and are generated using a component of the kernel called the smallstepper.

The two functions performed to support the RPC mechanism are the generation of the unique numbers by the smallstepper, and the location of objects by the operation switch.

The function of the Operation Switch is to forward an invocation request to the appropriate type manager at the local or a remote node. These calls may be from a type manager or from the network driver. Each call contains the following information:

1. The extended UID of the object against which the call is invoked.
2. The extended UID of the process invoking the operation.
3. The extended UID of the principal on whose behalf the operation is being invoked.
4. The operation and a set of parameters.

The Operation Switch uses the host hint field of the target object's extended UID to determine whether the object is on the host or not. If it is, it uses the type unique number of the object to direct the call to the proper type manager. If the object is on another host, the Operation Switch instructs the Network Handler to send the call to the other host.

1.3.1.2.3 Unique Identifier Generation

Unique identifiers generation in Zeus is an integral part of the system. It is the only information that is generated in a distributed manner. The type, instance and version fields of an extended UID are unique identifiers. Each of these unique identifiers consists of three fields: the host identifier of the host at which they were generated, the incarnation number, and the sequence number within an incarnation.

In a system where no failures can occur, each host will generate a monotonically increasing sequence of unique identifiers. If we permit failures, but stipulate that every host in the system has stable storage, then each host will store the next incarnation number and as soon as it starts it will retrieve this number and write to stable storage the next incarnation number, thus even though some part of a range of sequence numbers may not be generated, the hosts will generate a monotonically increasing sequence of unique numbers.

If we remove the assumption of stable storage on all hosts in the system, then hosts in the system can be divided into two classes: those that possess stable storage and those that do not. Each host in the system has a process called the smallstepper which issues unique identifiers for a given incarnation number. Each host with stable storage in addition to the smallstepper has a process called the largestepper which together with the other largesteppers in the system generates new incarnation numbers. The algorithm used to do this is specified in section 1.4.2.1.

1.3.1.2.4 Network Handler

This component provides a simple datagram level of transport mechanism between different kernels. It interfaces with the Operation Switch. The invocation requests for remote nodes are handed over by the Operation Switch to the Network Handler, which has the responsibility for delivering it to the Operation Switch at the destination host. Similarly the response messages are returned from the server to the invoker by the network handler via the Operation Switch.

1.3.2 User Visible Functions

As mentioned previously, Zeus is a set of type managers whose members may potentially change dynamically as type managers are created, deleted, and modified. There is, however, a subset of type managers called the System type manager which perform the essential services provided by the kernel of a conventional operating system. In this section, the type managers for these system types are defined. The following are the System type managers which exist at each node in the system.

(1) Type-type manager
(2) Process/Transaction Manager
(3) Principal and Authentication Manager
(4) Symbolic Name Manager
(5) Program Manager
(6) Message Manager

The functions provided by these type managers along with their structures are described below. Each of these type managers is considered as an object of distributed type; an instance of each of these type managers resides at every node. The distributed type managers for a given type function cooperatively to provide the abstraction of a single system-wide type manager.

1.3.2.1 Type-Type Manager

The definitions of new type managers are introduced in the system by using the mechanisms supported by a system-wide object called the Type-type manager; thus, the Type-Type manager implements functions to create, alter, delete and replicate Type Managers. The definition of the Type-Type object given here is an adaptation and extension of the Type-Type concepts originating in the HYDRA [WULF81] operating system. The facilities provided by the Type-type manager include an explicit command on where to locate copies of a type manager.

1.3.2.1.1 Organization of Type/Type Manager

The basic operations of create, modify and delete are summarized below:

1. Create-type : Creates a new type manager on a set of hosts.
2. Install-type : Makes a copy of an existing type manager on a set of hosts.

<u>Create-Type</u>

The Create-Type operation creates a new type definition and installs it on a set of hosts. In order to create a type, the creator of the type must supply a set of extended UIDs for program objects. These are the data structure and operation specifications of the new type. In addition, the user

may supply extended UIDs of programs that will implement consistency control, concurrency control and other components of the infrastructure of the type manager.

The Create-Type call has the following structure:

```
create-type (numprog :  in integer; programs :  in template;
             numhosts :  in integer; hostnames :  in hostspec;
             typUID :  out extUID; status :  out result);
    where
         numprog - number of component programs the user
                   has supplied.
         programs - an array of extended UIDs of the
                   programs. This could be a more
                   sophisticated structure, like a
                   template with slots for special
                   functions.
         numhosts - number of hosts on which the type
                   manager must be installed.
         hostnames - the names of the hosts on which the
                   new type manager must be installed.
         typUID - is the UID of the new type.
         status - the result of the call.
```

Install-Type

The install operation is the initiation of a new type manager on one or more hosts in the system, therefore, its invocation should logically follow the create-type operation. The Type/Type manager copy who processes the call will obtain a data base record of the type and then send it to each of the hosts specified. Those hosts will install the type manager provided they do not have it. The call to install-type is

```
install-type (typeUID :  in extendedUID; numhosts :  in integer;
              hostnames :  in hostspec; status :  out result);
    where
         typeUID - UID of the type
         numhosts - count of hosts on which the new type
                   manager is installed

         hostnames - the names of the hosts
         status - the result of the call
```

1.3.2.2 Process/Transaction Manager

Processes and transactions are active objects in the system through which a user carries out operations in the system. Transactions are atomic processes, i.e. they have an "all or nothing" property. The transaction facility with its atomic property provides a powerful mechanism for reliable operations. A transaction either commits or aborts on termination, and if it aborts then no trace of its execution is left. On the commitment of a transaction, all updates made by it are permanent.

We require that a process must invoke a transaction in order to modify permanent shared objects in the system. The changes to an object are recorded as new versions of the object. New versions of the object are committed to becoming permanent at the end of a successful completion of the commit protocol among the invoked transaction, the invoking process, and the type managers of the modified objects. Uncommitted versions are discarded on explicit abort commands issued by the transaction process or on timeout due to inactivity.

Processes and transactions can establish recovery points by checkpointing. Such points are used for the purpose of rollback and restart of a process or transaction. Checkpointing is the selective saving of versions of process or transaction objects. Note that with the above scheme for checkpointing, only the state of the process (or transaction) object is saved; the states of objects modified by that process are not saved in the checkpoint. This approach may create problems for error recovery since not all state changes of the process are recorded with the checkpoint. However, one must remember that all updates made within a transaction to permanent objects via their type managers are saved on the stable storage as uncommitted versions.

The Process/Transaction Manager also supports nesting of transactions; such nested transactions can execute concurrently with the parent transactions. The nested transaction facility provides the users mechanisms to introduce concurrency within a transaction. The commitment of a nested transaction is dependent on the commitment of the parent transaction.

1.3.2.2.1 Process Manager Functions

This section describes those functions of the Process/Transaction Manager which are designed from the point of view of recovery. This section is divided into two parts. The first part (numbers 1-7) introduces the allowable operations from this subset of functions on process type objects and the second part (numbers 8-12) introduces those on transaction type objects.

1. Create_Process (Program_UID, [list of Data_UID, host_ID, Expected_Time]): Returns Process_UID

The Create_Process operation requires at least a program UID. It creates a process, assigns it a UID of process_UID type and starts running the process. The process will be aborted if it does not terminate within the period of time specified by the Expected_Time Parameter. If this parameter is not given, then unlimited time is assigned to the process. The Receive_msg operation (described in communication management) can be invoked to check whether a DONE message, which indicates the termination of a process or transaction, has been received or not. The purpose of this operation is to wait for the DONE message in the block_wait state. Then the status of the process can be acquired by initiating the Process_Status function (discussed later).

Parameter Description:
Program_UID: The UID of the object which contains the program.

List of Data_UID: None, one or more UID(s) of the object(s) which contains the required Data.

Host_ID: The ID of the host where the new created process is to reside. This parameter is not required if the created process is to be on the same host as its parent process.

Expected_Time: The maximum period of time that is expected for the created process to terminate. Unlimited amount of time is assigned to the process if no value is given for this parameter.

Value Returned:
Process_UID --> Indicates the success of the operation.
NULL_UID --> Indicates the failure of the operation.

2. Delete_Process ([Process_UID |Transaction_UID]):Returns (0,1)

The Delete_Process operation deletes the specified process or transaction object regardless of its current status.

Parameter Description:
Process_UID |Transaction_UID: The UID of a process or transaction, requested to be deleted. This parameter is not required if a process or transaction wants to delete itself.

Value Returned:
1-->Indicates the success of the operation.
0-->Indicates the failure of the operation.

3. Process_Status (Process_UID |Transaction_UID):
Returns (Non_existent, Running, Aborted,
Time_Out, Completed, Suspended, Crashed)

The Process_Status operation returns the current state of the process having Process_UID or of the process which executes the transaction having the Transaction_UID.

Parameter Description:

(Process |Transaction)_UID: The UID of the process whose current status is requested; OR the UID of the transaction whose executing process current status is requested.

Value Returned:
Non_existent --> Indicates a process or transaction UID which does not exist.
Running --> Indicates the process is in the Running state.
Time_Out --> Indicates the Time_Out occurs before the process terminates.
Aborted --> Indicates the process has been aborted.
Completed --> Indicates the process is completed.
Suspended --> Indicates the process is in the

suspended state.
Crashed --> Indicates that the host where the process resides has crashed.

4. Establish_Recovery_Point ([ERP]): Returns (0,1)

The ERP operation saves the current state of the process or transaction object in stable storage. Successive calls to this function increments the RP_Num by one and stores the state of the process or transaction at the time of call in stable storage. The updated RP_Num indicates the latest recovery point number within the context in which it is called. The first recovery point for each process or transaction has the value of zero and it is established automatically when a process or transaction starts its execution.

Value Returned:
1 --> Indicates the success of the operation.
0 --> Indicates the failure of the operation.

5. Discard_Recovery_Point ([Process_UID, RP_NUM1, RP_NUM2]):
Returns: (successful, non-existent_RP,
access_control_violation, non-existent_UID)

The DRP operation discards all recovery points whose RP_num is equal to and includes specified recovery points between. If just one recovery point is to be discarded the RP_Num2 is not required, and the last recovery point is discarded if none of the RP_Nums is specified.

Parameter Description:

Process_UID: The UID of process whose recovery point(s) are discarded. If a process object wants to discard any of its own recovery points, then this parameter is not required.

RP_NUM1: The RP_NUM1 specifies the recovery point to be discarded. If more than one recovery point is to be discarded, the RP_NUM1 indicates the starting recovery point number.

RP_NUM2: This parameter is needed if more than one recovery point is to be discarded. It gives the recovery point number of the last recovery point to be discarded.

Value Returned:
Successful-->Indicates the success of the operation

Non_existent_RP-->Given recovery point(s) is out of range or does not exist.

Access_violation-->Indicates the process is not authorized to discard the recovery point of the process given by the Process_UID.

Non_existent_UID-->The process with the given UID does not exist.

6. Rollback ([RP_NUM]): Returns: (0,1)

The Rollback operation within a process restores the state of all the local objects to their values which they possessed at the time the RP_num was established. The changes on global objects, which have been made by transactions within that process, remain permanent if the transactions performing those are committed; otherwise, they are restored to their values that they possessed at the time the RP_num was established.

The Rollback operation within a transaction restores the state of all the local and global objects to their values that they had at the time the RP_num was established.

Parameter Description:

RP_NUM: It specifies the recovery point number to which the transaction or process state is being rolled back. If the parameter is not specified, the process or transaction rolls back to its last recovery point.

Value_Returned

1-->Indicates the success of the operation.
0-->Indicates the failure of the operation.

7. Last_Recovery_Point ([Process|Transaction)_UID]):
Returns: RP_NUM

The Last_Recovery_Point operation is invoked to find the recovery point number of the last recovery point of process or transaction identified by the process or transaction UID.

Parameter Description:

(Process|Transaction)_UID: The UID of the process whose last recovery point is requested.

Value Returned:

RP_NUM $\geq$ 0 --> Specifies the last recovery point.
RP_NUM $\leq$ 0 --> Indicates the failure of the operations, that can be due to access violation or invalid Process_UID.

Following is the description of user visible functions for transaction type objects which are designed in detail.

8. Begin_Transaction ([T])

The Begin_Transaction command creates a new transaction and by executing this statement system establishes the first recovery point (RP_NUM=0) and generates a UID of Transaction_UID type for that transaction. If parameter T is given, then it contains the Transaction_UID, after this statement is executed.

9. End_Transaction

The END_Transaction operation is the commit point for the transaction; thus, for an outermost transaction, execution of the End_Transaction statement means permanence of all updates made within this transaction.

10. Create_Transaction (Program_UID, [list of Data_UID|, host_ID, Expected_Time]):

Returns: Transaction_UID

The create_Transaction operation requires at least a program UID. It creates a transaction, assigns it a UID of transaction_UID type, and starts running the transaction. The transaction will be aborted if it does not terminate within the period of time specified by the Expected_Time parameter. The user can acquire the status of the transaction by initiating a Transaction_status operation (discussed later).

Parameter Description:

Program_UID: The UID of the object contains the program.

list of Data_UID: None, one or more UID(s) for the objects(s) contains the required Data.

host_ID: The ID for the host where the new created transaction is to reside. This parameter is not required if created transaction is to be on the same host as its parent transaction.

Expected_Time: The maximum period of time that is expected for the transaction to terminate. Unlimited time is given to the transaction if this parameter is not specified.

Value Returned:

Transaction_UID-->Indicates the success of the operation
NULL_UID-->Indicates the failure of the operation

11. Commit_Transaction ([Transaction_UID, ERP]):
Return (Non_existent_UID, Environment_Violation, Successful, Unsuccessful)

The Commit_Transaction operation makes all the updates which have been performed by a transaction permanent. This function can be called only by a non-transaction process that has created some concurrent transaction (by executing the Create_Transaction function). Therefore, no nested transaction is committed by calling this function; the commitment of a nested transaction occurs when its parent transaction executes its End_Transaction command. The execution of this command for a nested transaction is still valid; however, such an invocation of this command will not commit a nested transaction.

Parameter Description:

Transaction_UID: The UID of the transaction to be committed.

ERP: If this parameter is true, then the execution of this command establishes a recovery point and stores the state of all local variables and global objects at commit time.

Value Returned:
- Successful --> Indicates the success of operation.
- Non_existent_UID --> Indicates the transaction with given UID does not exist.
- Environment_violation --> Indicates the transaction has attempted to commit a transaction which is not within its execution context.
- Unsuccessful --> Indicates the transaction is not in a state that can commit.

12. Abort ([Process_UID |Transaction_UID]):
Returns (Non_existent, access_control_violation, successful)

The Abort operation terminates the execution of the current block and restores the state of the local variables and global objects to their values before the beginning of transaction and continues execution with the statement immediately following the End_Transaction statement of the aborted transaction. If Abort command is used within a process, it terminates the process.

Parameter Description:
(Process|Transaction)_UID: The UID of process or transaction that is to be aborted. If no value is given for this parameter the process or transaction is aborted itself.

Value Returned:

Non_existent --> Indicates an invalid UID which does not exist.

Access_control_violation --> The process (transaction) is not authorized to abort the given process or transaction.

Successful --> Indicates the success of the operation.

Establish_Recovery_Point, Discard_Recovery_Point, Last_Recovery_Point are exactly the same as those for processes. For deleting a transaction, Delete_Process can be used by giving Transaction_UID.

1.3.2.3 Principal and Authentication Manager

The object protection system in Zeus depends on the ability of the individual type managers to identify any process which requests an operation be performed. In addition, the Type Managers need to be able to determine the

ultimate initiator of the action which resulted in such an invocation request. We call these initiators of actions principals. Principals are permanent objects in Zeus and they are the only objects which carry the authority to perform computations involving other objects. When a new process is created, it is "owned" by a single principal and it retains this principal association throughout its lifetime.

The two fundamental problems of the protection system, authentication and authorization, both involve principal objects and the association of processes to principals. The problem of authorization, that is determining on whose behalf a given process is currently working, is a fairly simple matter since each process is always working for a single principal only. When a process invokes an operation on a type manager, the information regarding its UID and principal association is transported onto the virtual machine of the target type manager. In this way, the principal which owns a particular process is always known by any type manager on which it makes invocation requests. In addition, since process identifiers are transported to and from type manager machines by system code, a process is unable to forge its own principal association to gain access to objects that the process' real principal is not authorized to access.

During login, a user is first asked to identify himself by giving his unique principal symbolic name. The login process (also called the Authentication Manager) tries to find a principal object containing the same symbolic name. The principal object contains all the pertinent information about that user. The user's password is stored with the principal object, allowing the Authentication Manager to perform necessary authentication checks. Two other pieces of information regarding the user are maintained within the principal data object. One is the unique identifier (UID) of the user's symbolic name context, which is described in the next section. The other is the UID of the command interpreter or shell program of the logged-in principal.

Since the authentication manager must find a principal object given only its symbolic name, it follows that this name must be unique. In order to make it convenient for unique names to be assigned to principals, Zeus has the concept of a working group (WG). Working groups are used to form a strict hierarchy of principal names. This hierarchy of names is similar to that used in the Multics system. They contain members which may be either principals or other working groups. The root working group has a null name and is called the null working group. The unique name of a principal or working group is formed by concatenating the name of the principal or WG with the names of all of its containing working groups. This hierarchical structure also forms the basis for other symbolic names in the system.

1.3.2.3.1 User_Visible Authentication Functions

A full definition of the principal object data structure is as follows:

```
TYPE principal IS
     RECORD
        name :  principal-name; -- Unique user name
        passw :  password; -- Login password
```

```
        cntxt :  context; -- User's name context
        shell :  program; -- User's shell program
        plist :  process-list; -- Active process list
    END RECORD;
```

The principal type manager defines the following functions on principal objects:

1. create (nm : in principal-name; pw : in password;
   shell : in program; cntxt : in context)
   --> (return-code)

   Creates a new instance of a principal object and initializes its data fields with the principal name, password, shell program, and context given as parameters. The possible return codes are (1) new principal UID or (2) error in creating new principal.

2. delete (pid : in principal-UID) --> (return-code)

   Destroys a principal object given its UID. The possible return codes are (1) operation successful or (2) principal-not-found.

3. get-context (pid : in principal-UID) --> (return-code)

   Returns the context UID for a principal or principal-not-found error code.

4. lookup (nm : in principal-name) --> (return-code)

   Finds the UID of a principal given its unique symbolic name. The possible return codes are (1) the principal UID or (2) principal-not-found error code.

5. authenticate (pid : in principal-UID; pw : in password)
   --> (return-code)

   Returns true if the given password matches the one stored in the principal object and returns false otherwise.

6. get-pw (pid : in principal-UID) --> (return-code)

   Returns the password associated with a given principal. The use of this function should, of course, be administratively limited.

7. new-pw (pid : in principal-UID; pw : in password)
   --> (return-code)

   Replaces the password associated with a given principal. Again, access to this function should be controlled.

8. get-shell (pid : in principal-UID)
   --> (return-code)

   Returns the UID of the user's shell program.

9. new-shell (pid : in principal-UID; shell : in program)
   --> (return-code)

   Replaces the principal's shell program.

10. attach (pid : in principal-UID; proc : in process-UID)
    Adds the UID of a process to the active process list stored in the given principal.

11. detach (pid : in principal-UID; proc : in process)
    Deletes the UID of a process from the active list stored in the given principal.

12. get-procs (pid : in principal-UID) --> (return-code)
    Return a list of currently active processes working on behalf of the given principal.

Since the authentication manager must find a principal object given only its symbolic name, it follows that this name must be unique. However, it is usually convenient for a user to use his own name as his principal name and, obviously, this does not always identify him uniquely (i.e., John Smith). In order to make it somewhat more convenient for unique names to be assigned to principals, Zeus has the concept of a working group. Working group objects are maintained by the working group manager. Working groups are used to form a strict hierarchy of principal names. They contain members which may be either principals or other working groups. The root working group has a null name and is called the null working group. The unique name of a principal or working group is formed by concatenating the name of the principal or WG with the names of all of its containing working groups.

The data structure of a working group object is as follows:

```
TYPE working-group IS
     RECORD
          nm : wg-name; -- Symbolic name
          pid-list : principal-list; -- Member principals
          wgid-list : wg-list; -- Member WGs
     END RECORD;
```

The operations defined for working group objects are the following:

1. add-principal (nm : in principal-name; pw : in password;
   shell: in program; cntxt: in context) --> (return-code)
   Create a new principal object (by calling the principal type manager) and install this principal as a new member of the working group indicated by part of the given principal symbolic name. Initialize the data fields of the principal object with the given parameters.

2. add-working-group (nm : in working-group-name) --> (return-code)
   Create a new working group object and install it as a new member of the working group indicated by part of the given symbolic name.

3. delete (wgid : in working-group-UID) --> (return-code)
   Delete the given working group object only if it currently contains no members. The possible return

codes are (1) operation successful, (2) working group not found, or (3) working group not empty.

4. princ-membs (wgid : in working-group-UID) --> (return-code)
   Return a list of principal members of the given working group.

5. wg-membs (wgid : in working-group-UID) --> (return-code)
   Return a list of the working members of the given working group.

6. lookup (nm : in working-group-name) --> (return-code)
   Lookup a working group by its symbolic name. The possible return codes are (1) the working group UID or (2) working group not found error.

In some circumstances, it will be desirable to create abritrary, non-hierarchical groups of users. For example, a group of users may agree to share access to some set of objects. To accomplish this, the principal identifier of each of the group members could be added to the access list of each of the shared objects. A more flexible and efficient solution would be to define a group object or "security group" as a list of member principals and to place the single UID of this group on the objects' access lists. This would shorten the access lists in addition to allowing member principals to be added and deleted easily. Security groups are not implemented as separate composite objects in Zeus. Instead, the ATTACH and DETACH operations defined for principals are used to obtain the desired result of allowing arbitrary user associations to be formed. Normally, the only user authorized to perform the ATTACH operation on a given principal is the system principal and the log-in process. However, certain principals, called aggregate principals, may contain many other users on the access list of the ATTACH operation. These users are called "members" of the aggregate principal and they are allowed to create and ATTACH processes to the aggregate. Such an association of users thus functions as a security group but, since aggregates appear as ordinary principals to the individual type managers, the Zeus design is both more uniform and more efficient.

1.3.2.4 Symbolic Name Manager

The symbolic name manager (SNM) is one of the autonomous, distributed processes which comprise the Zeus operating system design. This type manager is solely responsible for the creation, deletion, modification, and management of instances of the symbolic name context type. The structure of these objects are not known outside the symbolic name type manager (SNM), storage is not allocated for them outside the SNM, and their values may not be assigned or checked for equality by any process other than the SNM. In short, all access to symbolic name contexts is completely controlled by the SNM.

1.3.2.4.1 Symbolic Name Contexts

Symbolic name contexts are the only supplied means within Zeus for a user to define and use symbolic (non-numeric) names for other system objects. Fundamentally, a context is a single- valued functional mapping from user-supplied symbolic names to system defined unique identifiers (UIDs). As such, the context plays a very important role for the user/principal since the system itself deals only with the bit string UID which is decidedly non-mnemonic but nonetheless efficient as a system name.

Each principal which has permission to logon to Zeus is associated with at least one context object which contains his own private names for objects with which he may interact. This arrangement allows a relative symbolic name space for principals in the sense that different users will, in general, have different symbolic names for identical system objects (that is, objects with the same UID). Such a relative scheme is very efficient in a distributed environment such as Zeus since names must remain unique only within a single context. This eliminates the need for a central (and therefore vulnerable) naming authority or a complicated hierarchical scheme as in other distributed systems. The primary shortcoming of this relative naming scheme is that it makes it difficult for two or more principals to become aware of a common shared object by passing its symbolic name. The principals are likely to have different names for the shared object and thus will be unable to find "common ground" on which to agree on a single name.

This type of object sharing by different user processes appears to be rather rare so that the overhead and added complexity of a hierarchical system may not be necessary. Instead, Zeus provides the required "common ground" by providing an aboslute symbolic naming scheme for the principals themselves. Each object will then have at least one non-ambiguous name with which principals may refer to it. This name is obtained by concatenating the principal's system unique name with that principal's relative name for the object to be shared.

Context objects may also be used to implement the aboslute naming scheme for principal objects. This is done by providing a single additional context object which contains the name => UID mappings for all the principals currently authorized to use the system. The symbolic name of this context (contexts may have symbolic names just like any other object) should be well known throughout the system. This can be accomplished by initializing each of the principal's contexts with the symbolic name of the unique user name context.

Due to the essential nature of a principal's symbolic name context, it will be desirable to provide the capability to define highly reliable contexts which are likely to survive or perhaps continue to be available in the presence of certain types of errors. The standard mechanism in Zeus for providing such reliable objects is object replication. Context objects are the first significant example of object replication which have been encountered in the detailed design.

Despite the requirement to provide distributed, replicated context objects in Zeus, the consistency requirements of the operations which will be defined for contexts are relatively simple. Straightforward read/write consistency control will be used in the SNM since the five operations defined therein appear semantically identical to four write operations and one read operation. In general, of course, some object types may have more complicated semantics for their operations such that two or more update operations turn out to be compatible since they modify mutually exclusive parts of an object instance. This, however, is not the case with symbolic name contexts.

1.3.2.4.2 User's View of the SNM

For user processes in the Zeus system, the SNM (and in fact all other type managers as well) appears as a single autonomous server process which defines an abstract data object type and a set of operations which access and/or modify instances of that type. The actual structure of the defined data type, as previously mentioned, is not known by the user process. In addition, the physical host boundaries (indeed most physical characteristics of the system) are hidden from the user process so that the physical network of hosts appears as a logical network of connected user and type manager processes.

The user-visible features of a type manager are completely defined by the set of operations which it provides. The symbolic name type manager provides five such operations on context objects; CREATE context, DELETE context, ADD name, REMOVE name, and LOOKUP name. The operations have the obvious semantics. CREATE and DELETE operate on whole contexts while ADD, REMOVE, and LOOKUP modify and access the individual name/UID pairs in an existing context object. The LOOKUP operation is the only read- only operation while the others all cause at least part of the context object to be modified in some way. The reason that ADD and REMOVE are considered to be modifying the entire context object rather than only a single entry in it is that the most likely implementation of context is as a hash table. This means that any modification of the pointer structure within a hash container to insert or remove a name/UID pair would cause the pointers to become temporarily inconsistent thus requiring that the entire table be made unavailable.

1.3.2.4.3 SNM Functions

The Symbolic Name Manager provides functions to create, maintain, and destroy contexts and symbolic name to UID entries within contexts. The following are operations that affect the whole of some context object.

create () --> (context-id, return-code)
  Creates a context object with the denoted access list. The possible return-codes are (1) true and (2) false.

delete (context-id : in out context) --> (return-code)
  Deletes a context object. The possible return-codes are (1) deleted and (2) non-existent context.

The three operations, ADD, REMOVE, and LOOKUP are provided to maintain the symbolic names to object UID mapping. Of these three, only the LOOKUP operation is read-only while the other two cause a name to UID mapping within the context to be modified.

```
add (name : in symbolic-name; object-id : in UID;
    context-id · in context) --> (return-code)
          Create a symbolic-name / object UID entry in the
          given context. The possible return codes are (1)
          successful and (2) non-existent context.
lookup (name : in symbolic-name; context-id : in context;
       name-id : out UID) --> (return-code)
          Find the symbolic-name in the given context and
          return the object UID associated with that name.
          The possible return codes are (1) successful, (2)
          non-existent context and (3) name is not found.
remove (name : in symbolic-name; context-id : in context)
          --> (return-code)
          Delete the symbolic-name / object UID entry from
          the specified context. The possible return-codes
          are (1) successful and (2) non-existent context.
```

1.3.2.5 Program Type Manager

The Program Type Manager is the repository of both program text and object code. Program text is defined to be a text object that compiles correctly; thus, the creation of a program object requires the user to supply the Program Manager with a correct program or a separately compilable unit of a program. The Program type manager, in addition to its function as a repository, acts as a builder of programs; thus, a user can call upon the program type manager to build a new program from some specified components. This linking function of the Program Type Manager is useful to the system to build new user types. A program object is defined to be a collection of versions of a single program. The criteria for retaining program versions in the system are defined by the users.

1.3.2.5.1 Program Object Functions

There are four functions that are necessary for program objects. These are create, delete, construct, and gettext. The function construct is the linker that builds a composite program object from a set of components. The other functions are those necessary to maintain the program repository. Some of the functions may require auxiliary functions which will be described at the end.

1. Create

Create is invoked when a new program object or a new version of a program object is to be installed in the program manager. This function is supplied with the text file for the new program by the user. The function compiles the text file and installs the text file as a program unit provided the

compilation was successful. The function installs the program unit with a unique version number which it returns to the caller. It must be noted here that a user cannot create a new program version in place by taking an existing version, modifying it within the Program Type Manager and then creating a new program unit. Instead, the user must first get the program text, modify it using some function and then use the create function to install it as a program unit. A call to the create function is

```
    create (textobj :  in text; progname :  in out ext UID;
            listing :  out text; status :  out result)
  where
    textobj - program text
    progname - .is null if the text object is to be installed as
                a new program
               .has a null version field if this is a new
                version of an existing program object
    listing - a text that lists the compiled text and errors
    status - returns the result of the operation and can be
               .successful
               .no entry in the directory
               .cannot be installed in the directory
```

The function does not return a value in progname if the compilation is not successful.

2. Delete

The delete function is invoked to delete a program object or a specific version of the program object. The deletion of a version of a program implies that its empty slot in the array is filled by moving the other versions (if any) to fill the slot.

```
    delete (progname :  in extUID; status :  out result);
  where
    progname - is the extended UID of the program.  If the
               version unique number is not present then the
               delete will delete the program while the presence
               of a version unique number deletes a single
               version.
    status - is the return code from the delete function.
```

3. Construct

The construct function is used to build a program object from a set of program objects. Like the create function, it can be used to create a new program object or a new program version. This function can be used to build new type managers. The construct function will use the function getcode to obtain the code of each of the components and it will use a linker to build the new program object version. The new version or extended UID will be returned to the caller. The call to construct is:

```
    construct (prog_count :  in integer; type_name :  in template;
               status :  out result; progname :  in out extUID;
               listing :  out text);
```

where
prog_count - number of program extended UIDs in the array type_name
type_name - array of program object extended UIDs. These are the components of the new program object.
status - return flag from the routine.
progname - if null a new program object is created and its extended UID is returned; if the version field is null then a new version is created.
listing - a listing of the results of the operation.

4. Gettext

This function copies into a user parameter the text of a program unit. The text is for the version specified or for the latest version. This function may be used to copy a program to an editor and then modify it. The call for gettext is

```
gettext (progname : in extUID; textobj : out text; status :
        out result);
```

where
progname - extended UID of the program object
textobj - text of the program object
status - result of the operation

1.3.2.6 Message Type Manager

Within the Zeus Operating System, the Message Type Manager (MTM) provides messages as a means of inter-process communication in either a synchronous or an asynchronous fashion. A message that is transferred from one process to another is viewed as an object upon which operations are performed to effect this inter- process communication. The operations are send_msg, receive_msg and msg_status. The send operation creates a message object and initiates the transfer of the object from the sender to the intended receiver. The receive operation completes the transfer when the object's message content is returned to the receiver. The sender or receiver can determine the status of a message object by performing a msg_status operation.

1.3.2.6.1 Reliability of Message Objects

At the time a message is created, the sender can specify the reliability class for that message. The reliability class of a message reflects its availability to the receiver in the face of one or more host failures in the network. At the low end of reliability there are volatile message objects that disappear upon host failure (if the object resides on the failed host). At the high end of reliability stable message objects have a replication factor of n where n is the number of hosts in the network. The four reliability classes are volatile, non- volatile, resilient and stable.

1.3.2.6.2 Scope of Inter-process Communication

Inter-process message communication may occur between processes that are local to a host, or remote. In either case the send and receive operations are performed on the MTM local to the host of the calling process. Any remote communication that might be performed to effect the respective operation is carried out between MTMs and is unseen by the caller. Figures 1-2 and 1-3 depict the flow of information in local and remote inter- process communication respectively.

1.3.2.6.3 Message Operations from the User's Viewpoint

The messages are sent by invoking a send_msg operation with the specification of the message content, the list of receivers, the reliability class of the message, and whether or not the message operation is to be performed synchronously or asynchronously. If the send is a synchronous send, the sender is delayed for the shorter of the sender timeout value or for the time it takes to route all of the copies of the message to any remote hosts. The routing is a function of the reliability class and the receiver list. When a send is asynchronous, the sender is blocked only the time it takes to create the message and notify the receivers of its existence. As a result of the send operation, the message unique identifier is returned. If the message status is determined at a later point by the sender, it is this unique identifier that is passed as a part of the msg_status call.

In order to receive a message, the receiver invokes a receive_msg operation and specifies which processes to receive a message from, and whether or not the operation is asynchronous or synchronous. This means no wait if there is no message available in the asynchronous case and a wait in the synchronous case until a message is available or until a timeout occurs. An additional parameter allows the user to further qualify which message is received. The qualification may indicate that either the most recent message be received, or the oldest, or the first since a host failure.

The msg_status operation returns to the caller the current status of the message relative to its receipt by each of the intended receivers of the message. Possible statuses are received, not_received, unavailable and message non_existent.

Following the descriptions of the MTM_Interface components Send_Msg, Receive_Msg, Msg_Status, and Message_Operations.

1.3.2.6.4 Send_Msg

The user interface to perform message manipulations is a procedural one. Calls are made to routines named send_msg, receive_msg and msg_status that are part of a Message_Interface package within the process space of the user. The following are the procedural interfaces between the user process and the Message_Interface routines.

```
PROCEDURE send_msg (msg_vars:  IN MTM_type.parm_list;
                send_to_list:  IN MTM_type.xid_list;
                      option:  IN MTM_type.wait_no_wait;
                     timeout:  IN POSITIVE;
           reliability_class:  IN MTM_type.rel_classes;
                      msg_id:  OUT kernel.xtnded_uid;
               return_status:  OUT MTM_type.msg_opn_return);
```

Parm_list is a record that describes the variables that compose a message. Some convention will be made between the compiler(s) of a host machine and the send_msg procedure as to the actual record description of parm_list.

The send_to_list is a linked list of the intended receivers of the message. A broadcast of a message is indicated when the send_to_list is composed of a single star, "*".

The process has the option of waiting for acknowledgements that the message has been sent to every receiver or not waiting for the acknowledgements. This is specified by "wait" or "no_wait" as the value of the option If the option is wait, a timeout value must be specified which is the maximum time that the sender is willing to wait for the acknowledgements.

The reliability_class for a message object may be volatile, non-volatile, resilient, or stable. A volatile message object is one with the least likelihood of being available if some failure occurs because it is a single copy object in memory. A stable message object has the greatest likelihood of being available because a copy of the message exists on each host, and is thus a replicated object. Non_volatile and resilient message objects are more reliable than volatile objects and less reliable than stable objects. The number of message copies created during a send operation and their storage medium will be varied during performance analysis to determine what combinations provide the maximum amount of reliability and efficiency. One major difference between non_volatile and resilient is that non_volatile objects have no recovery operations performed for them upon failure, but resilient objects do.

The msg_id is a unique identifier for the message that is returned after the message is sent. This identifier may be used in a msg_status call to determine the state of the message regarding its receipt.

The return_status contains the result of the send operation and may be completed or timed_out. The not_completed status will at a later time be expanded into a group of possible error return values according to the fault that caused the operation to fail.

1.3.2.6.5 Receive_Msg

The complement to send_msg operation is the receive_msg operation that a process invoked to receive a message that is available.

```
PROCEDURE receive_msg (msg_vars:  IN MTM_type.parm_list;
               receive_from_list:  IN MTM_type.xid_list;
                     wait_option:  IN MTM_type.wait_no_wait;
                which_msg_option:  IN receive_option;
                         timeout:  IN POSITIVE;
                          msg_id:  OUT kernel.xtnded_uid;
                       sender_id:  OUT kernel.xtnded_uid;
                   return_status:  OUT MTM_type.msg_opn_return);
```

Msg_vars are the variables into which a received message is placed.

The receive_from_list indicates which process the receiver is willing to receive from. It may be a linked list of process extended uids, or a star (*) which indicates a willingness to receive from any process.

The wait_option may have the values of either wait or no_wait where wait will cause the receiver to wait a finite amount of time for a message to arrive, the wait time being indicated by timeout.

The which_msg_option may be either most_recent, oldest, or first_after_failure. This gives the receiver flexibility in receiving messages.

The msg_id contains the extended uid of the just received message.

The sender_id contains the extended uid of the process that sent the message.

The return_status may be completed or timed_out.

1.3.2.6.6 Msg_Status

The current status of any particular send message operation may be determined with the msg_status operation.

```
PROCEDURE msg_status (msg_id:  IN kernel.xtnded_uid;
            return_statuses:  OUT MTM_type.msg_opn_return_list);
```

The msg_id is the extended uid of the message for which a status query is being made.

The return_status record is a linked list of process_id/status pairs., That is, one status is returned for each intended receiver process. The possible return statuses are received, not_received, unavailable (i.e., status not known), and non_existent.

In order to receive a message, the receiver invokes a receive_msg operation and specifies which processes to receive a message from, and whether or not the operation is asynchronous or synchronous. This means no wait if there is no message available in the asynchronous case and a wait in the synchronous case until a message is available or until a timeout occurs. An additional parameter allows the user to further qualify which message is received. The

qualification may indicate that either the most recent message be received, or the oldest, or the first since a host failure.

The msg_status operation returns to the caller the current status of the message relative to its receipt by each of the intended receivers of the message. Possible statuses are received, not_received, unavailable and message non_existent.

1.4 OVERVIEW OF THE DETAILED DESIGNS

1.4.1 Zeus System Design

The design of the Zeus system is basically the design of the kernel and system type manager functions described in Section 1.3. A detailed design of each system type manager is presented in this part of the guidebook. An important part of the Zeus design is the design of a generic object manager; this design defines the protocols executed by an object manager with the Process/Transaction Manager to ensure reliable operations and recovery in the system. The detailed designs presented here were carried out in CSDL and Ada. It is suggested that an interested reader should refer to the Concurrent System Definition Language (CSDL) manual to understand the CSDL designs. The designs of the Process/Transaction Manager and the generic Type Manager are presented in CSDL. The detailed designs of the Kernel and two system type mangers, namely, the Symbolic Name Manager and the Message Type Manager, are presented in Ada. The following sections of this chapter present an overview of these designs. During this effort we have not designed the Authentication Manager and the Program Type Manager in detail.

1.4.2 Kernel Design

The overall structure of the kernel functions is given in Figures 1-4, 1-5, 1-6. The structure of each function is described next. Each remote procedure can, as was pointed out earlier, be broken up into four messages. Each message must be packetized at its source and re-assembled at its sink. There are four levels in the RPC mechanism. The first is a set of kernel procedures that form the kernel interface for the RPC mechanism. These procedures are invoked to send calls and obtain responses. In addition they differentiate between local and remote objects thus performing the function of the operation switch.

The next level consists of one call handler per object type and is the repository of the state of all outgoing and incoming calls. Each call handler is invoked by the kernel procedures from the level above it and the network handler processes from the level below. The network handler consists of three processes, the send driver, the receive driver and the network tranceiver. Together the send and receive drivers implement the link level protocol. This includes the processing of acknowledgements (positive and negative) at the packet level. The send and receive drivers interact with the network

tranceiver to send and receive data packets to and from the network. The network tranceiver interacts with the UID generation task and the net controller in addition to the send and receive drivers.

Object storage and retrieval has a very simple structure. Each object type has a simple object directory and a stable object directory. Associated with each directory are sets of tasks, one set for each operation to be performed. These tasks are called request handlers. The kernel interface procedures first ask the directory to perform some operation. The directory initiates the task and returns to the kernel procedure a pointer to the request handler assigned to the task. The kernel procedure then asks the request handler for the result. The directories contain pointers to all objects of a given type. The directories always reside in main memory and directory storage and retrieval has not been considered in the design. Storage of the directories would involve setting up a directory structure. Request handlers are allocated to each directory based on the designer's estimate of how often operations on a particular object type will be invoked and how long those operations will take. Secondary device space is allocated on an object basis and simple and stable storage have free storage managers. The two kinds of storage are managed separately since they will be kept on separate devices.

1.4.2.1 UID Generation Protocol

The third major function of the kernel is to generate unique identifiers for objects. A unique identifier consists of a host field, an instance field and a sequence field. The instance field is generated collectively by all the hosts to the system. For each value of the instance field the UID monitor process will generate a range of UIDs whose cardinality equals that of the sequence field. The generation of a new instance number is accomplished by the UID generation process of the kernel. Additionally, the UID generation process participates in a roll call computation to keep track of active hosts, and a host restart computation to permit a new or restarting host to join the set of active hosts.

All the UID generation processes on the active hosts in a cluster participate in the generation of a new instance field. The algorithm for UID generation is based on the principle that all the active hosts in a cluster have a dynamically maintained linear order at all times. When all the active hosts in the cluster have received a new instance number request from one of their members they each set a timer in the process UID timer and wait. If a host's timer expires before that host receives a response to the request it will broadcast a response to the request. This message contains the new instance field value and the responding host's position in the linear order. When a host receives a response to a request (either its own or that of a host with a lower position in the linear order) it accepts the new instance value and subtracts the responding hosts position in the linear order from its own. Thus each active host receives the new instance number. Further, if some host in the linear order has failed, the other hosts ranked lower than it will have their position moved higher in the order. This will over time result in reliable hosts reaching the highest positions in the linear order.

In Zeus, the system processes for UID generation belong to a set of smallsteppers S or a set of largesteppers L. These processes reside on hosts which are interconnected by a broadcast network that uses a contention protocol. All hosts (i=1..n) must possess an instance si ∈ S. Some hosts (j=1..m), called stable storage hosts, possess an instance lj ∈ L. The union S U L forms the set of processes.

The object of the system is to generate unique identifiers (UIDs) on request from other processes in the system. Requests for a new UID on host i are directed to si. Each si has a limited range of UIDs it can supply. si obtains ranges of UIDs from a specially denoted lj which is called the elector (le). Each range is called an incarnation. A single largestepper must respond to a request for a new incarnation. The new incarnation is adopted by all the processes si and lj active in the system.

The algorithm to obtain new incarnations must be resilient enough to:

1) Recover from host failures which result in the le pro cess becoming unavailable.
2) Recover from host failures which result in some li process becoming unavailable.
3) Insure that the incarnation numbers are generated in a monotonically increasing sequence.

The algorithm assumes the following about the underlying system:

1) An errorless, loss-free transmission medium.
2) Delivery of messages in a FIFO manner.
3) Continuous availability of the broadcast medium.

Among the sets S and L the following assumptions hold:

1) An elector holds his position from the instance of his nomination until the host on which he resides fails.
2) A smallstepper si is considered to be idle if it has sent out an incarnation request and is awaiting a reply.
3) At any time the active largesteppers li ∈ L possess a unique priority number pi from 0..|L|-1. At any time the active largestepper with pi equal to zero is by default the elector.
4) There is a global constant t possessed by each largestepper. This constant t is the maximum time taken by any largestepper to receive and broadcast a message.

The algorithm requires the following messages:

1) A request message which is sent by some si which requests a new incarnation number.
2) A reply message which is sent by the elector le to give the new incarnation numbeb. This message also holds the priority number of the largestepper who generated the message.

The algorithm is based on the notion of a priority ordering. Upon the receipt of a request message broadcast by some smallstepper si each largestepper will wait for a time period equal to pi times t for a reply message.

If it receives no reply message in that time, the largestepper broadcasts the reply message with the new incarnation number and reduces its priority number pi to zero.

If the largestepper receives a reply message before the time period expires, it STOPS its timer, installs the new incarnation number and subtracts the value of the priority number in the message from its own priority number. The smallsteppers si receive the reply message, too; thus their incarnation number is updated.

The largestepper with pi = 0 will, if active, time-out immediately and send out the reply message; thus, the largestepper with pi = 0 is the elector at any time.

If the largestepper with pi = 0 is inactive then the largestepper with pi = 1 will time-out, send the reply message and then become the largestepper with pi = 0.

Thus, the first reply message with priority number k implies that all largesteppers with priority numbers from 0 to k-1 are inactive. The remaining largesteppers on receiving the reply message decreme their priority number by k and reset their timers.

The above statements will insure that the incarnation number will be generated as quickly as possible provided there is an active largestepper. However, if two or more smallsteppers request an incarnation number simultaneously, then multiple incarnation numbers will be generated.

1.3.2.6.6.1 The Time Constant t

A crucial element of this algorithm is the value of the time-out constant t. This constant must be large enough to insure that the largestepper process with priority pi can be scheduled on its host and broadcast its message on the network before the largestepper with priority pi+1 can time-out and broadcast its message.

We are assuming that this time is variable but bounded and that t is the least upper bound. This assumption is trivially true in the case of token passing networks. For CSMA networks (e.g., Ethernet) this time may not be bounded due to network contention and multiple collisions. However, boundedness could be arranged in a CSMA network by giving priority to the largestepper's reply by reducing its retransmission interval below that for other network traffic.

Within the host careful specification of process priorities can help insure an upper bound on the processing time required to handle a incarnation number or priority request.

1.3.2.6.6.2 Introducing a New Largestepper to the Network

The new largesteppgr must be given the current incarnation number and a priority number. We define active largesteppers to be those that possess the current incarnation number and a priority number. While incarnation numbers will be identical on all the active largesteppers, the priority numbers may not be continuous because of failures in largestepper processes. One of the aims of the algorithm below is to make all the priority numbers continuous, i.e., pi equal to 0..j when j+1 hosts are active. This is important because the values of pi must lie between 0 and m-1.

The algorithm to do this is:

1) The new largestepper broadcasts a priority request message (which contains its id) and waits for t times (L).

2) Each largestepper that was active on receipt of this message sets some variable newp to zero and sets a timer to t times pi.

3) While waiting for the timer to lapse, each active large stepper will receive priority reply messages whose value to pi is less than its own. For each of these messages, it will increment its value of newp. It will also ignore any other priority requests it receives during this time.

4) On the receipt of a message with a value of pi one less than its own or if the timer expires, the largestepper will broadcast a message with its incarnation number, its current value of pi and the id of the largestepper whose request it is serving. It will then set its value of pi to be the value of its variable newp. This ends the active largesteppers role in the algorithm.

5) If the largestepper that broadcasted a priority request receives priority reply messages that contain a host id other than its own, it will broadcast another priority request message after its time expires. Otherwise, given its large time constant, the new largestepper's time period will expire after all other largestepper time periods. At that timY KQwill simply set pi to be the count of messages it has received.

The above algorithm insures that the introduction of a new largestepper will set the priority numbers of the active largesteppers correctly. Further, multiple largesteppers that request a priority number simultaneously, will be given priority numbers correctly.

1.4.2.1.1 Reliability Issues in UID Generation

The above name generation scheme is not totally immune to failure. For example, let a Host A which contains a largestepper process be the last host to go down. Then this host contains the latest incarnation number. If the

system is now restarted with some other host B, then the incarnation numbers generated may be duplicates of those generated by host A before it crashed.

The above has the following implications:

1) For the names generated to be unique, some largestepper host must be active at all times.
2) If all the largestepper hosts crash then all of them must be brought up simultaneously.

It appears to us that the first restriction is reasonable enough. If all the largestepper hosts go down, then the new order for largesteppers could be obtained by selecting the largestepper with the largest incarnation number and priority equal to zero to be the new elector.

To periodically check for the active hosts the active hosts execute a roll call protocol. To do this they set the RECALL REMINDER process timer to be equal to some time delay plus a delay proportional to the host's position in the linear order. The first host whose RECALL REMINDER timer expires starts the roll call computation by sending out a start roll call computation message. The roll call computation proceeds with each active host setting two timers, rank timer and full timer, in the process rcall timer. The rank timer is proportional to the host's position in the linear order while the full timer is proportional to the maximum number of hosts in the cluster. When the rank timer expires, the host recalculates its own position in the linear order based on the number of messages it has received from hosts higher in the order. It then broadcasts its own 'I'm here' message on the network. In effect, the roll call computation is a periodic poll of active hosts.

Hosts that fail rejoin the system by initiating a distributed computation among the active hosts UID generation process. This computation is identical to the host roll call computation described earlier. This computation pre-empts the host roll call or UID generation computations. Thus the host restart computation is a poll of active hosts triggered by a new active host.

The messages sent by UID generation process take precedence over the remote procedure call messages. The UID generation monitor process when the latter exhausts the number's in the previous instance field's range. The UID monitor is invoked by the get_UID kernel procedure which in turn is invoked by the kernel users.

1.4.3 Process Manager Design

This section provides an informal overview of the design of the Process/Transaction Manager in the Zeus system. The formal definition of this architecture is given in Chapter 3. The presentation in this section is based on the fundamental notions in CSDL such as machines, sub-machines, and interfaces. In CSDL, a machine can communicate with its environment or with its submachines by using interface objects which are its public objects. Public objects are the objects visible to the outside world. The Process Manager is treated as a machine which has as its components some submachines. This machine contains application processes and some command processors as its submachines. This architecture is shown in Figure 1-7. The Process Manager machine interacts with the external world via some interface objects which are public objects for this machine. These interface objects include interfaces

to the secondary storage, memory manager, UID generator, and the Operation Switch.

This Process Manager machine has as its components some static as well as dynamic submachines. A machine is dynamically created from a POOL of the desired type of machine. Static submachines are those which are retained throughout the lifetime of Process Manager and dynamic ones are those which are created and destroyed dynamically. The static submachines of Process Manager are Timer, Router, and the PM_Database_Manager, whereas dynamic submachines are the application processes and the various command processors. The application processes are created from a pool of PROCESS machines. Each of the static and dynamic machines is connected to the parent machine Process Manager. The connections between Process Manager and static submachines are made in the beginning of PM Realization dictionary. PM is connected to Timer through a pair of mailboxes PM_TO_Timer and Timer_TO_PM. Similarly UID_GENERATOR is connected to PM through mailboxes PM_TO_UIDgen and UIDgen_TO_PM. The function of the router machine is to multiplex and de-multiplex the messages to (from) the Operation Switch from (to) various application processes and the command processors. The Router connects to PM via two connection paths. One path is reserved for application commands and responses between the PM and the Router, and the other path is reserved for control commands and responses between the PM and the Router. In addition, the PM connects to each process which are dynamic submachines via two paths. One path is reserved for application commands and responses, and the other is for control commands and responses. In CSDL, the dynamic machines can be declared as a pool of one type of machine and INDEX statement associates a name with each new machine created. This acts similar to the subscript of an array.

In response to application commands or requests from remote Process Managers, command processors are created. A command processor interfaces with the Process Manager's controller, and the PM_Database_Manager. An array of mailboxes called Command_Proc_Iface supports communication between the command processors and the PM_controller. Interfaces are also provided to the secondary storage and the primary memory manager by connecting the command processors to the shared ports called SS_Port and MM_Port. These shared port abstractions are supported by a machine_type called Port_Multiplexer. Each command processor directly interfaces with the Router machine that in turn provides communication path to the Operation Switch.

Besides connecting the PM machine to it submachines, we need to connect it to other independent machines to which it wants to talk to. Figure 1-7 shows three machines namely Stable Storage Manager, Memory Manager and Operation Switch to which the PM may communicate. The function of the Operation Switch is that of communication medium among different type managers. The PM sends or receives remote requests through the Operation Switch. These requests go through the Router machine which formats the messages in a proper way. Since a submachine cannot talk directly to an outside machine, the requests from the Router to the OS go through a pair of mailboxes namely Router_TO_OS and PM_TO_OS which are "bound" together to give the effect of just one mailbox. The other independent machines are connected to the PM as shown in Figure 1-7.

In addition to connecting PM machines to their submachines and other machines, some of the submachines of PM are connected to each other. The PROCESS submachines are connected to Router through a LIST of mailboxes, one for each PROCESS. This indirectly estblishes connections among each process and to the OS.

One other component of a machine is the CONTROLLER. This component processes the information gathered through public objects, local objects, etc. It acts as a driver for the machine. It generally consists of executable statements and procedures. In the PM, the PM Controller is a program which is inside a non-terminating loop. Similarly, the Router Controller is also a non-terminating loop which controls and executes functions of the Router. It is assumed that there are controllers in other submachines of the PM and independent machines outside PM, but they are not shown explicitly in the picture.

To support the dynamic creation/destruction of mailboxes for PROCESS machines in PM and Router, we declare the mailboxes as a LIST of records. Each record in the LIST consists of two fields: one of mailbox_type and the other of process_UID_type. In order to refer to any mailbox for a specific process, we use process_UID as the index. Since the PROCESS machines are dynamic, they are declared as a POOL of PROCESS machines to which a PROCESS machine can be added or deleted. The index to a PROCESS is associated with its PROCESS_UID to refer to that machine. A similar scheme is used to store the index of a command processor in its mailbox interface to the PM controller.

1.4.3.1 Command Processor

A command processor is created in response to either an Application request message or a request from another command processor. The general architecture of a command processor is illustrated in Figure 1-8.

There are eight different types of command processors. A command processor is dynamically created from a POOL of desired type. As shown in Figure 1-8, a command processor contains a static timer machine which generates the time out interrupts, also a command processor interfaces with the following machines:

Process_Manager	Through	Command_Proc_Iface
PM_Database_Manager	Through	PMDB_Iface
Operation_Switch	Through	OS_Iface
Stable_Storage	Through	SS_Iface
Memory_Manager	Through	MM_Iface
Parent_Process	Through	Parent_Iface
Child_Process(es)	Through	Descendent_Iface.

Since a command processor is a submachine within the PM it can not communicate directly to the independent machines outside the PM, namely Stable Storage, Memory Manager and Operation Switch. So the connections are provided through the SS_Multiplexer MM_Multiplexer and Router respectively.

If a command processor is created as a response to an application request the parent_Iface is to be bound to the PM_TO_Process mailbox of the applicant in the PM Controller; otherwise the parent_Iface is to be connected to one of the Descendent_Iface mailbox of the requester which itself is a command processor. In Chapter 3 two procedures Create_Command_Processor and Create_Appl_Server, are defined for creating the command processors. The procedure Create_Appl_Server is used for creating a command processor in response to a command invocation by an application process. In this case the PM_TO_Process mailbox for the caller process is bound to the Parent_Iface of the command processor. The second procedure, the Create_Command_Processor, is used to create a command processor in response to some command invocation by another command processor. In this case the Parent_Iface of the new command processor is connected to one of the Descendent_Ifaces of the invoker command processor.

1.4.3.2 Process Manager Database:

The PM database is the set of objects in the PM machine which contains necessary information about the ACTIVE processes at a local node. This database is also essential for PM to carry out its functions reliably. In other words, the database of Process Manager is a snapshot of the state of a local node of a particular instance of time. The PM database consists of the following LISTs:

1. LIST of Active_Process_Records: An Active_process_record may correspond to a process or a transaction. A transaction_record is the same as a process_record except for an additional field called transaction_status. A process_record contains the information about the UID, the LIST of processes that can access this process, priority of the process, process_state and time_out period. In CSDL, an Active_process_record is defined as discriminated union (variant record) of the process_record and transaction_record. The active_process_LIST is then declared as the LIST of Active_process_records.

2. LIST of Parent_Child_Info: This list records for each process or transaction the UID of its parent process. The Map_Field indicates its execution mode with respect to the parent process. The execution mode can be either sequential or concurrent. A transaction can be created as either a sequential or a concurrent process with respect to its parent. The Location field in this record indicates whether the parent is remote or local. The field Top_Level in this record is set true if the parent of a transaction process is a non-transaction process. A table called Descendent_Table records the UIDs of all of its children processes or transactions. Associated with each child is the list of the all descendent processes of that child and the UIDs of the objects modified by that child and its descendents. The RP_Child_Map field is used to find all descendents created after establishing a recovery point.

3. Directly_Modified_Object_List: This is a LIST of directly (in contrast to the objects modified by its children and grand-children) modified objects for each process and transaction. Each record in the LIST contains process/transaction_UID along with an array of modified objects. This

LIST is used for deleting versions of an object in case of an abort, rollback, or commit.

4. Current_Operation_List is maintained in the PMDB for recovery in case of system crash in the middle of some critical operation such as Rollback or Establish_Recovery_Point. This LIST contains records for each critical operation performed. Each record consists of a process_UID, operation name and its parameters.

5. PMDB_log_buffer is the LIST of all the modification done to PMDB. Each record of this LIST contains the operation performed which are ADD, DELETE, MODIFY and the information involved in these operations, for example, the record added or the record deleted, etc. This PMDB_log_buffer is periodically appended to a differential file on stable sotrage for recovery.

1.4.4 Type Manager Design

The Zeus system is a collection of object managers. Each manages an object of a specific type. Figure 1-9 depicts the architecture of a generic object manager which will be particularized at its creation time. This design has been developed independently but it can be easily integrated into the PM design presented in the Process/Transaction Management document. In general the function of an object manager is to perform operations on objects in response to operation invocation messages from client processes or teransactions. It enforces locking protocols and participates in commit protocols with PMs to ensure the atomicity of transaction. As shown in Figure 1-9 the object manager communicates with the PM and the Router through the TM_TO_PM and the TM_TO_Router interfaces respectively. The object manager contains a timer machine which is created when the object manager comes to existance and it is connected to that through TM_TO_Timer and Connection_TO_Timer. It also contains a POOL of servers. A server is a dynamic submachine of the object manager whose function is to perform the requested operation on an object. It is created in response to a request from a process/transaction and it is deleted after the operation is completed. These servers are functionally very similar to the command processors in the PM machine and they can communicate with the PM via the Servers_To_PM and the Servers_To_PM_PS.

The object manager database consist of the followings:

1) Objects: which is the set of objects and contains neccessary information about all the objects at that node and also the information about the transactions which are performing some operation on those objects. This information is essential for the object manager to carry out the commit protocol reliably and to preserve object consistency.
2) Queue : which maintains a list of all the requests that can not be processed immidately due the unavailability of the object (e.g., the object is locked in some incompatible mode.) The object manager must ensure that adding the request to the queue does not cause any deadlock.

The other component of the object manager machine is the CONTROLLER. This component is a program which is inside a non_terminating loop and processes the information which is gathered through TM_To_Router and the TM_To_Timer. This information can be either the rquests from the processes/transactions, received through TM_To_Router, or the timer interrupts.

1.4.5 Symbolic Name Manager Design

Symbolic name contexts are the only supplied means within Zeus for a user to define and use symbolic (non-numeric) names for other system objects. Fundamentally, a context is a single- valued functional mapping from user-supplied symbolic names to system defined unique identifiers (UIDs). As such, the context plays a very important role for the user/principal since the system itself deals only with the bit string UID which is decidedly non-mnemonic but nonetheless efficient as a system name.

Due to the essential nature of symbolic name contexts, it will be desirable to provide the capability to define highly reliable contexts which are likely to survive or perhaps continue to be available in the presence of certain types of errors. The standard mechanism in Zeus for providing such reliable objects is object replication; thus, the symbolic name manager must support relicated context objects.

The user-visible features of a type manager are completely defined by the set of operations which it provides. The symbolic name type manager provides five such operation on context objects; CREATE context, DELETE context, ADD name, REMOVE name, and LOOKUP name (see Figure 1-10). The operations have the obvious semantics. CREATE and DELETE operate on whole contexts while ADD, REMOVE, and LOOKUP modify and access the individual name/uid pair would cause the pointers to become temporarily inconsistent thus requiring that the entire table be made unavailable.

Internally, the symbolic name type manager consists of four major sections; the interface, the controller process, the operation processes, and the call handler. Figure 1-11 shows these four parts and indicates their relationship to one another.

The SNM interface is a collection of simple procedures, one per SNM operation, which runs as part of a user process and serves to interface it to the SNM proper. The interface is included by a user process which requires the services of the SNM and is found within the code library for the SNM. The interface knows about the parameters and protocols required by the type manager and it serves to hide these by providing the through the operation switch. This allows for a very consistent interface to the SNM in that all requests, local and remote, are handled in exactly the same way.

The remaining three parts of the symbolic name type manager together constitute the type manager proper. The first of these, the SNM controller, is respons'ble for fielding requests for operations on context

instances and then "spawning" operation processes to actually perform these operations. In order to do this, the controller contains within it a set of process queues, one per operation type, from which it schedules the process to be activated. Upon receiving and processing a request for an operation from a user process, the controller returns the identifier of the subordinate process. Subsequent dealings between the user and the operation process performing the work is handled by the user making calls directly to the operation processes thus bypassing the controller for efficiency.

The processes which the controller schedules to actually do the work for the operation requested by the user have a simple uniform structure. Each such process, upon initially awakening, accepts a call from the controller which starts the operation process running and also gives it a number which enables the controller to subsequently identify it. After accepting the intialization call, the operation process enters a loop in which it first accepts a "start" call to get all the required input parameters. The process then does whatever is neccessary to perform the requested operation. After the operation is completed, (or an error is detected), the operation process accepts a "done" call in which all the output parameters for the call are returned directly to the user process (actually his SNM interface). Finally, the operation process calls the controller to notify him that it has completed and that it may once again be added to the available process queue and eventually reassigned .

Since receiving a local or remote procedure call from a user requires a synchronous call to a kernel procedure, it would be undesirable to have the controller of the SNM make this call and thus be blocked until it completed. In addition, it is required that the controller present only a single interface to both local users and remote users. In order to facilitate this and to relieve the controller from having to do periodic calls to the kernel to poll for remote requests, a separate process called handler process is defined.

The call handler process simply calls the kernel to request the next remote procedure call and then blocks until this call is satisfied. Upon receiving a remote request, the call handler then "spawns" another small task which acts as the remote user's surrogate within the present host. The call handler then immediately calls the kernel to request more work. This surrogate user process takes care of packing and unpacking parameters and passing them to the kernel procedures and it uses the local controller interface in exactly the same way, as a local user would.

1.4.6 Message type manager Design

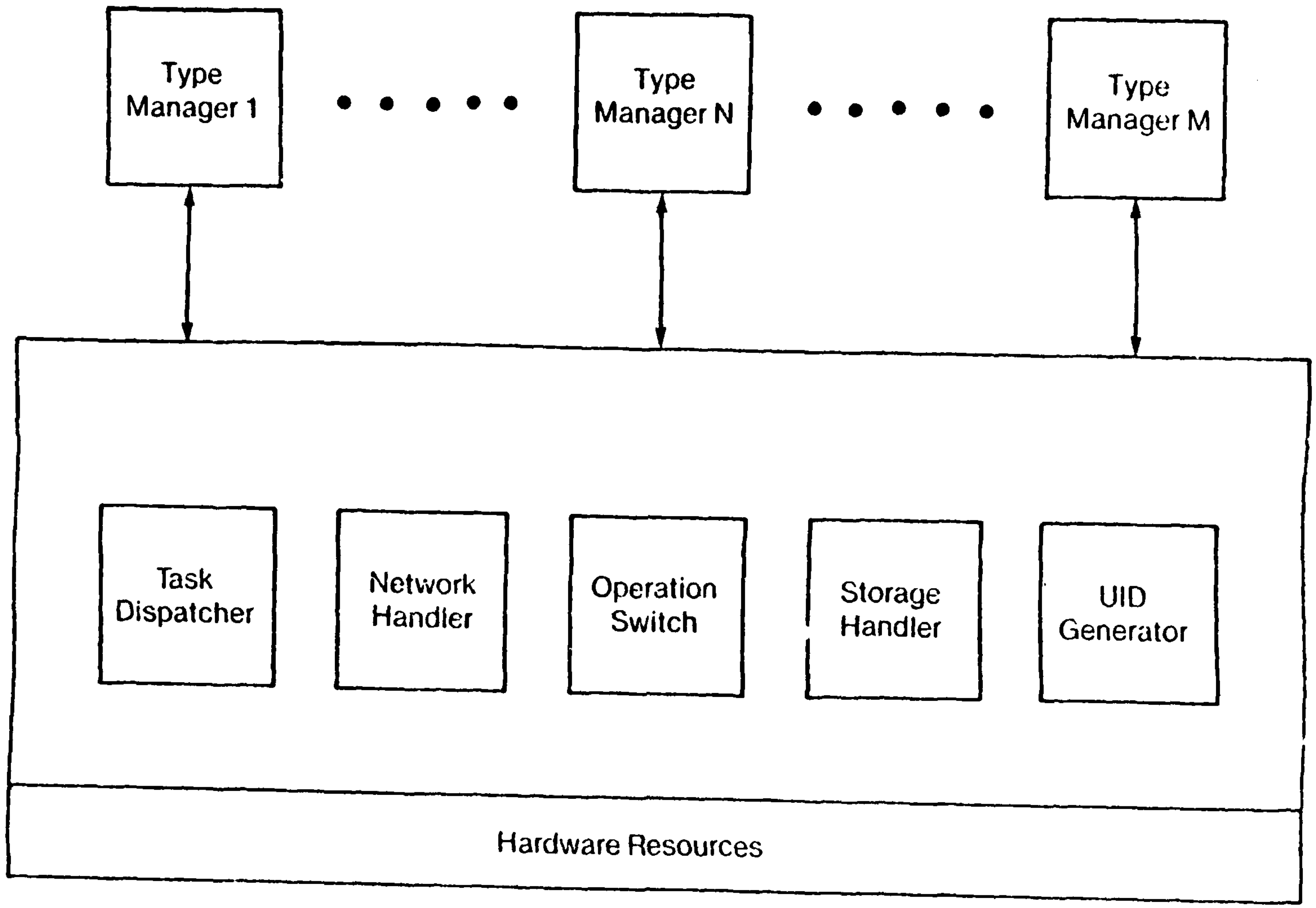
The MTM is replicated on hosts in the network wherever inter-process communication by messages is desired. The instances of the MTM are identical The composition of an MTM and its interface to the user is shown in Figure 1-12. User operation requests are made to the controller of the MTM which takes the appropriate action. A task to perform the requested operation is scheduled by the MTM_Controller from a pool of SEND, RECEIVE and MSG_STATUS tasks. (In the subsequent text, all capitalized words will refer to tasks of the MTM).

In sending a message, SEND calls the creation operation of MESSAGE OBJECT, which returns a message object, routes copies of the message object to remote hosts (as determined by the reliability class), and sends notices of availability to the intended receivers. A notice becomes an entry in the message queue for a receiver process. If the call is asynchronous, the message identifier is then returned to the sender. If the call is synchronous, SEND terminates but the sender remains blocked until SUPPORTER determines that some event has occurred and causes the sender to proceed (i.e., after a timeout or after all acknowledgements of copies sent are returned). When a sender is to be unblocked, the SUPPORTER schedules a WAKER task to bundle and route the appropriate response to the sender.

RECEIVE determines from PROCESS MESSAGE QUEUE (PMQ) whether or not a message is available that meets the specifications of the receiver. The PMQ manages all message queues for the processes of that host. It maintains the queues in stable storage. If there is a message available, it is returned to the receiver and RECEIVE terminates. If there is no message available and the call is asynchronous RECEIVE terminates and the receive continues without having received a message. In a synchronous call the sender remains blocked while SEND terminates and SUPPORTER performs the detection of the event to resume the receiver (either a timeout or an appropriate message arriving for the receiver). A WAKER task bundles a response and routes it to a waiting receiver.

The MSG_STATUS task returns the status of a message that is retrieved from a local copy of the message (if there is one), otherwise the status is returned from a remote copy.

One facet of the MTM not depicted in Figure 1-12 concerns the routine acknowledgement of events between MTMs regarding the routing of message copies. This is MTM_Controller to MTM_Controller communication via Kernel remote calls and responses. Some of the communications cause interactions with SUPPORTER (i.e. such an acknowledgement for a message copy sent to a remote host). Another interaction occurs when the PMQ is notified by its MTM_Controller of incoming notices of message availability for receivers on that host. When a copy of a message is required on a host where no such copy exists a request is made to a remote host that has a copy which causes a message copy to be routed to the requesting host. Such interactions occur between MTM_Controllers and are necessary for the smooth functioning of the operations of the MTM.



Structure of a Zeus Host
Figure 1-1

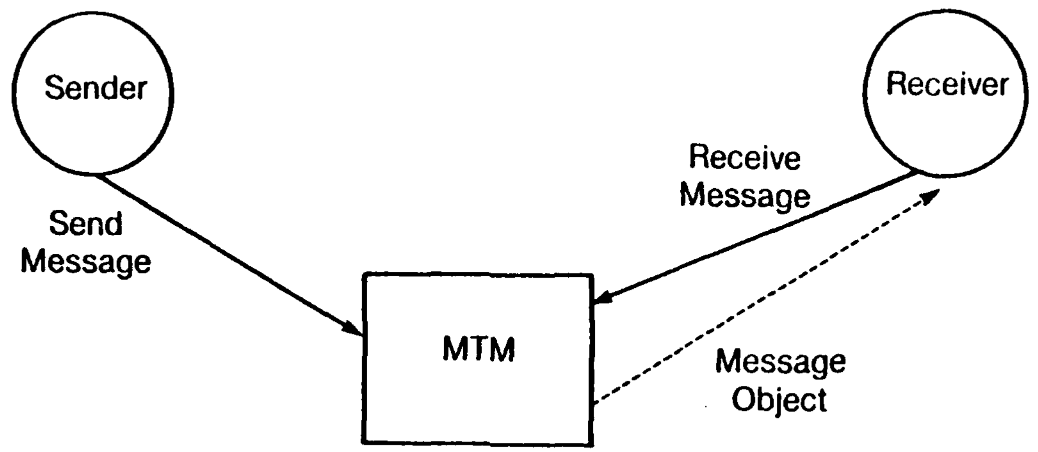


Figure 1-2. Local Message Communication

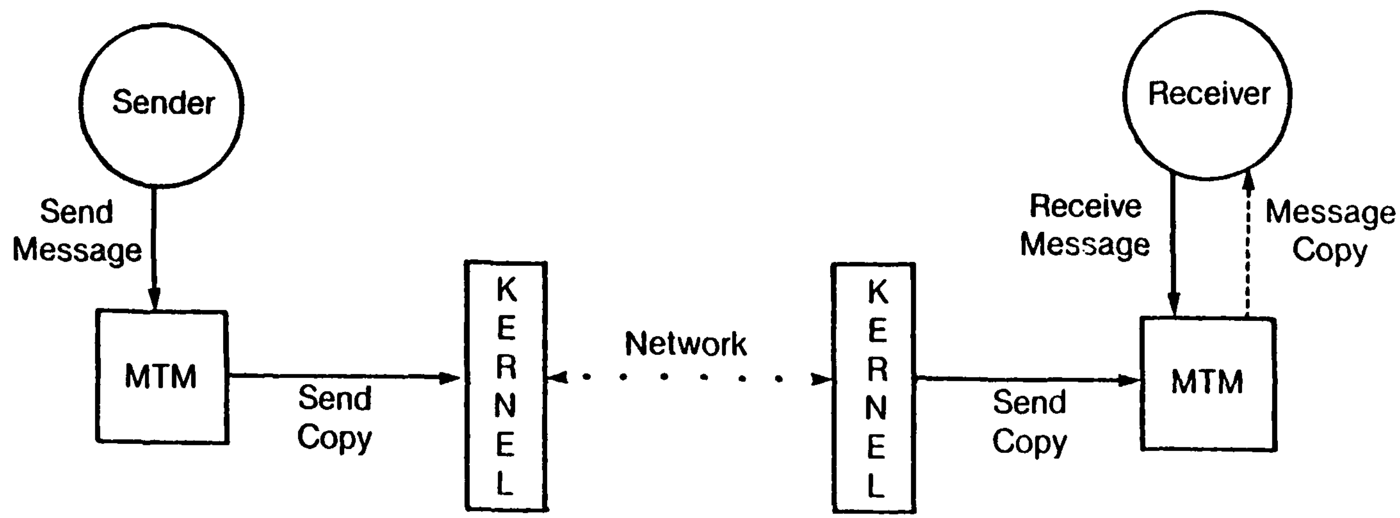
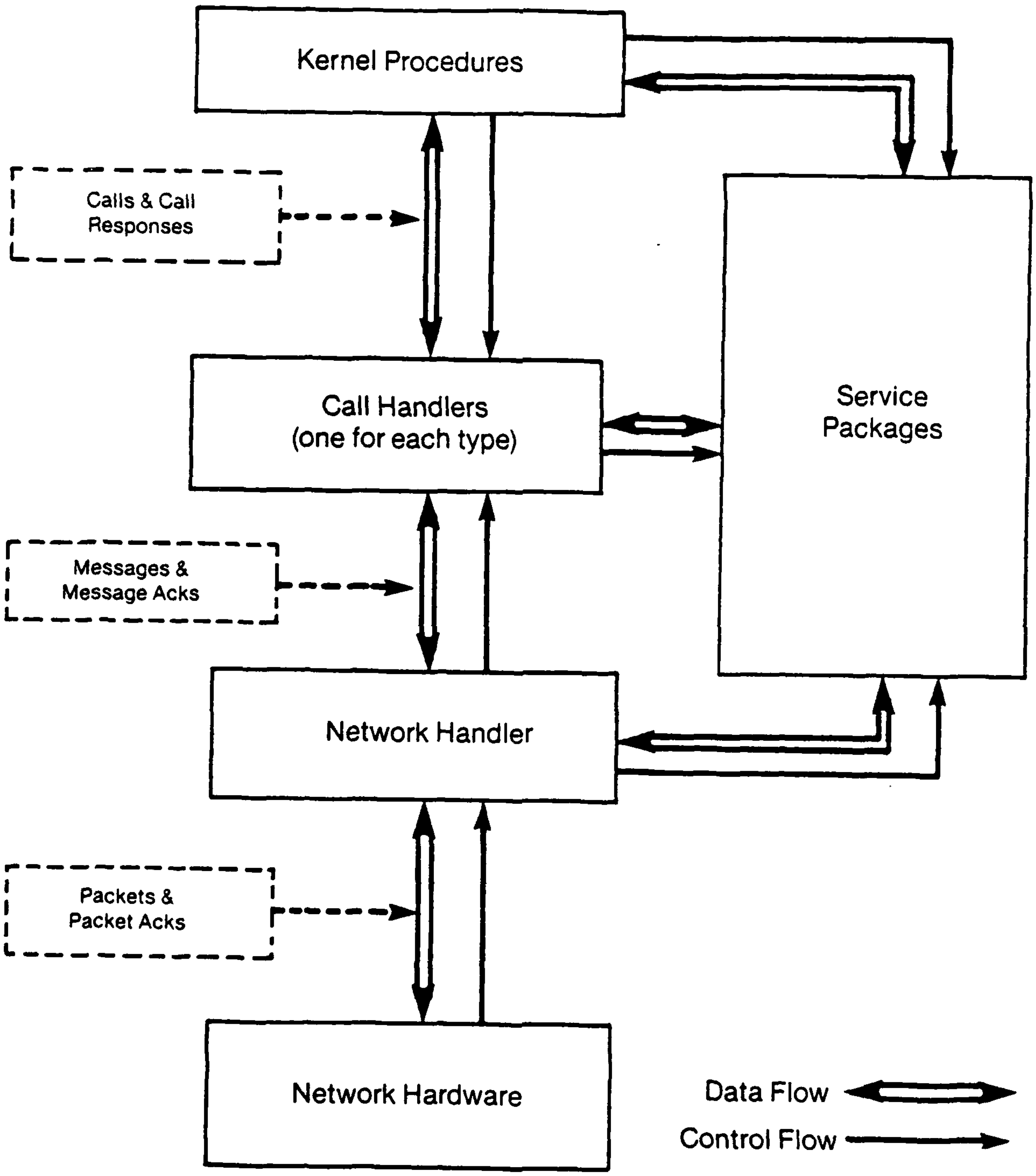


Figure 1-3. Remote Message Communication

RPC Major Components and Data Structures They Interchange



Major Components and Data Structures of RPC
FIGURE 1-4

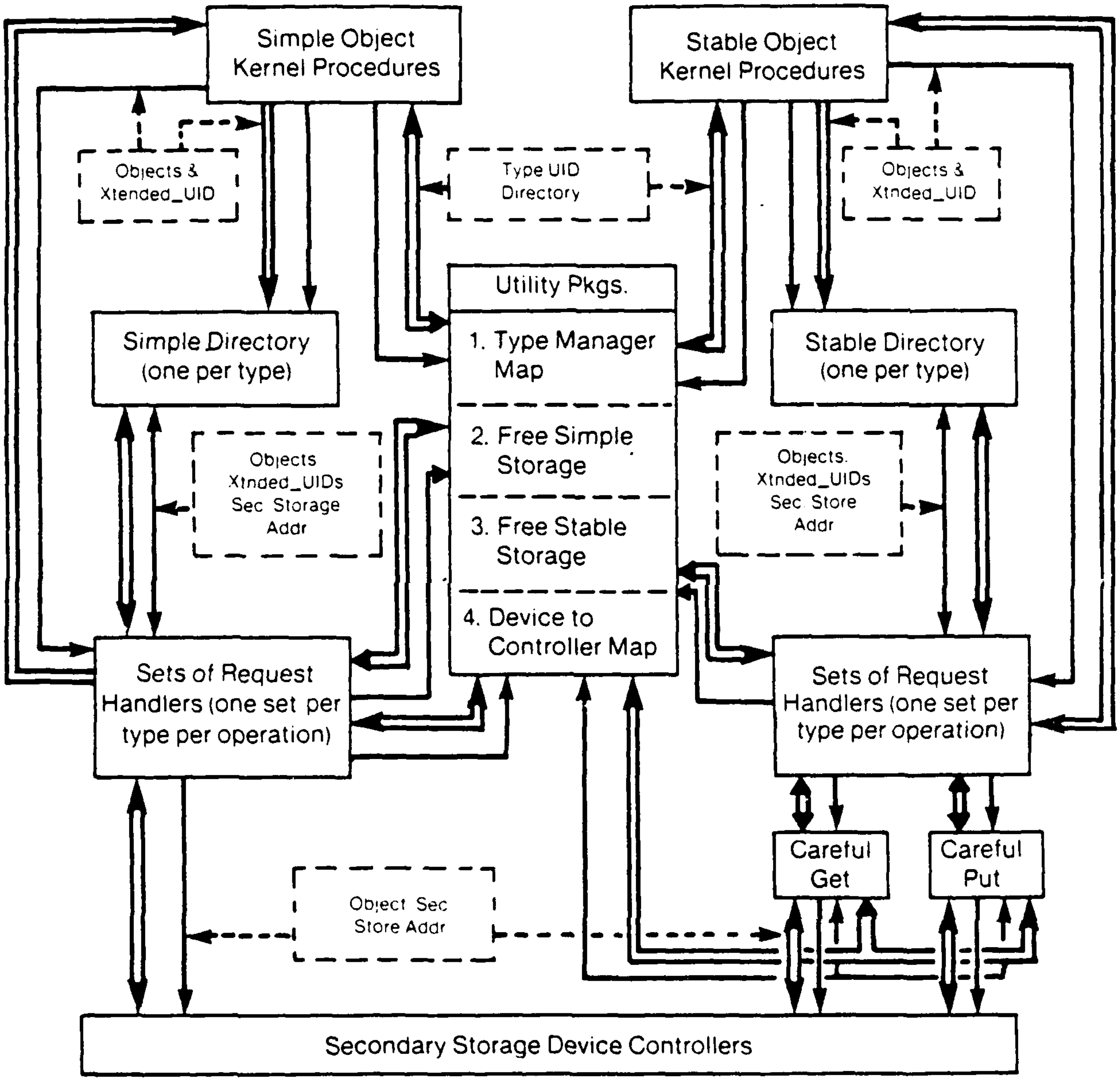
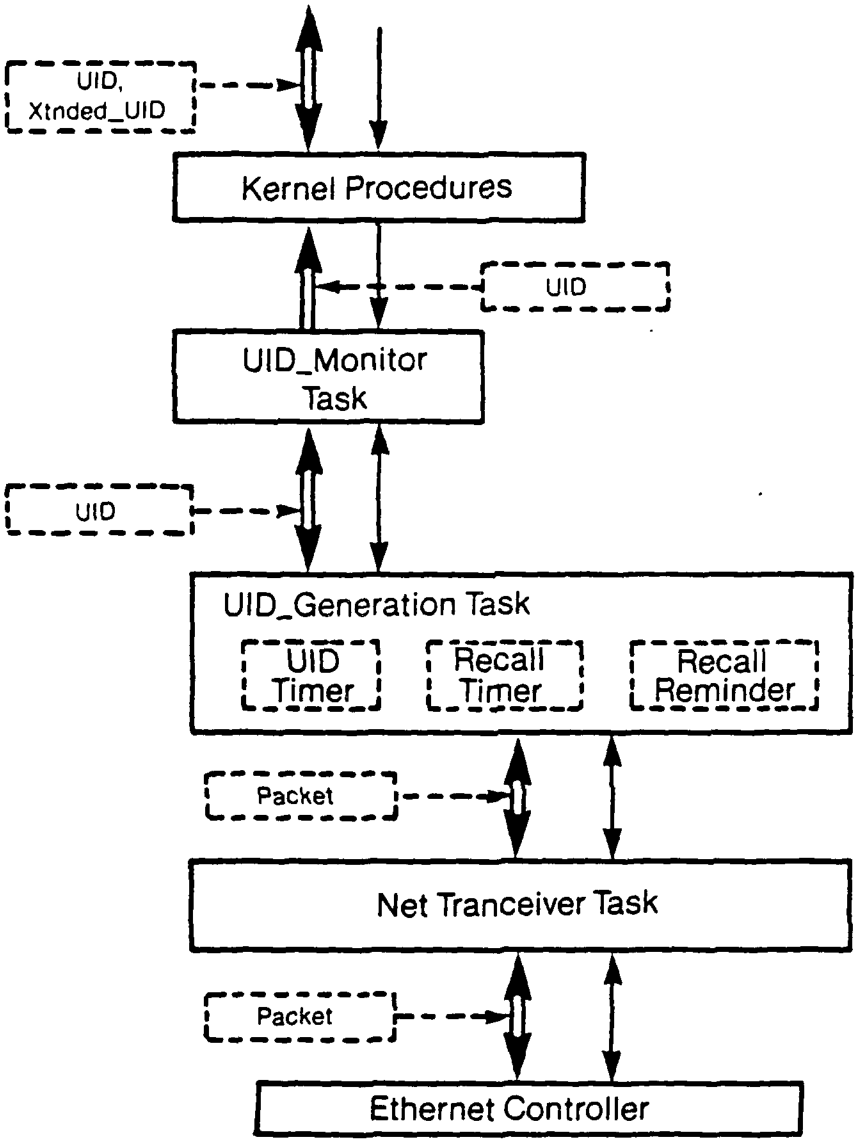


Figure 1-5
Object Storage and Retrieval:
Data and Control Flow



UID Generation and Site Recovery Architecture

FIGURE 1-6

PROCESS MANAGER ARCHITECTURE

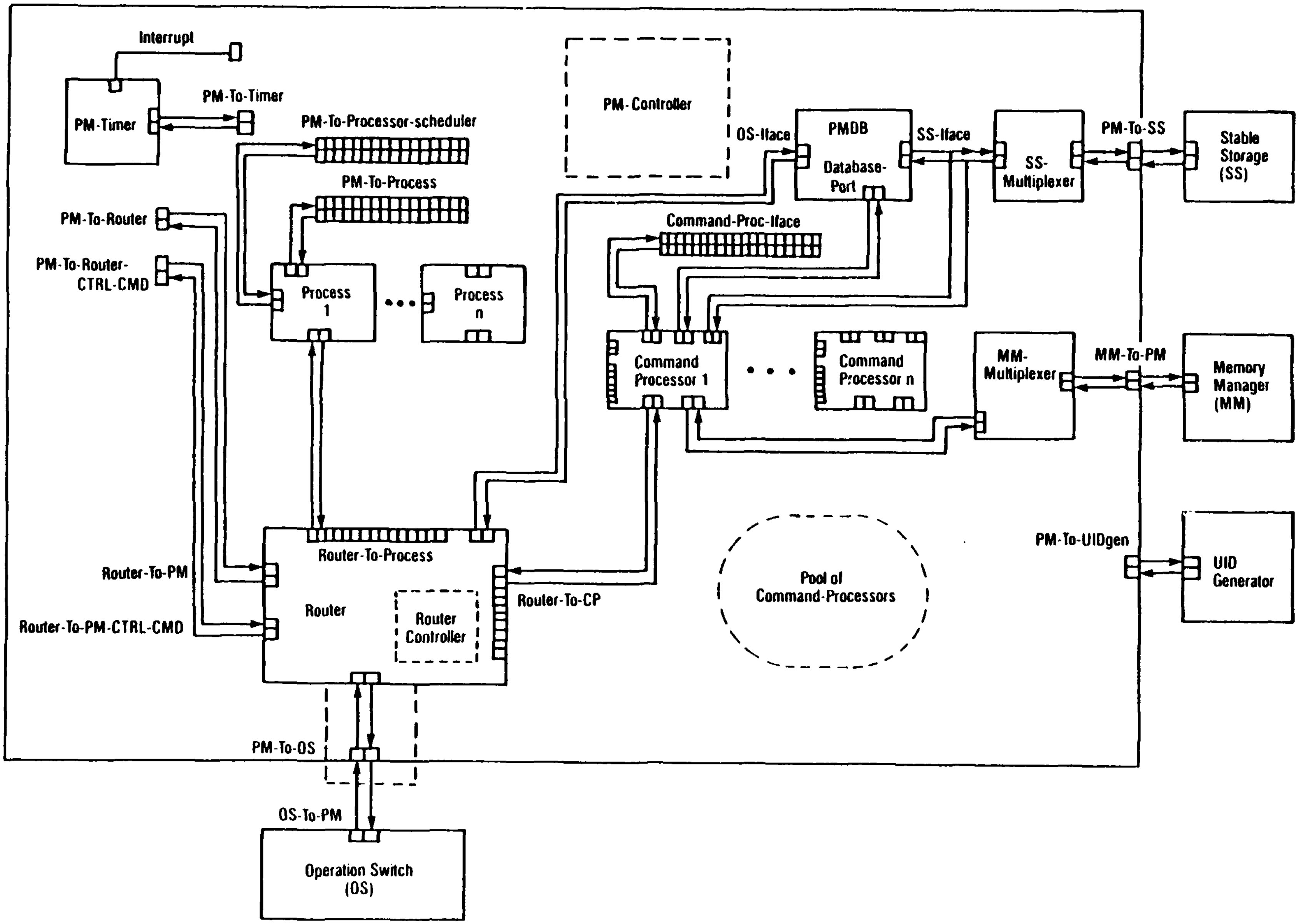


Figure 1-7

COMMAND PROCESSOR

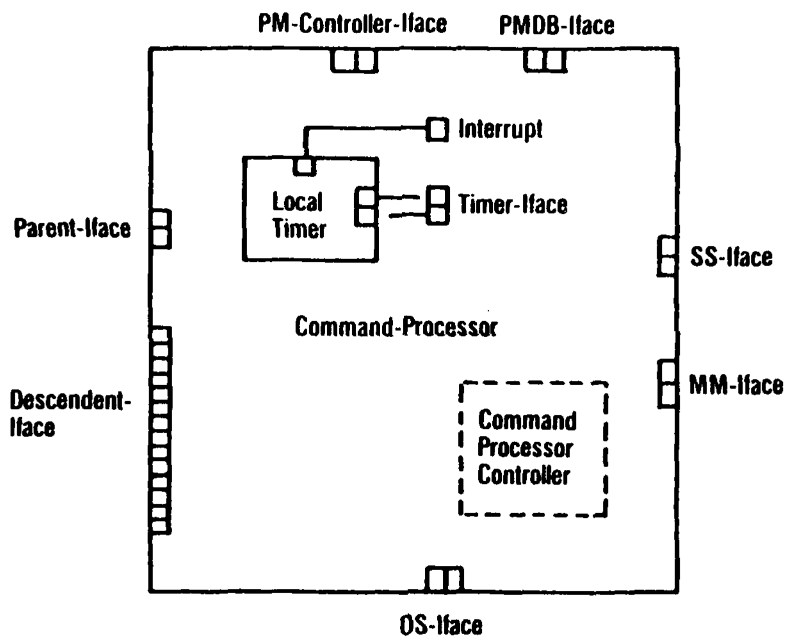


Figure 1-8

Type Manager Architecture

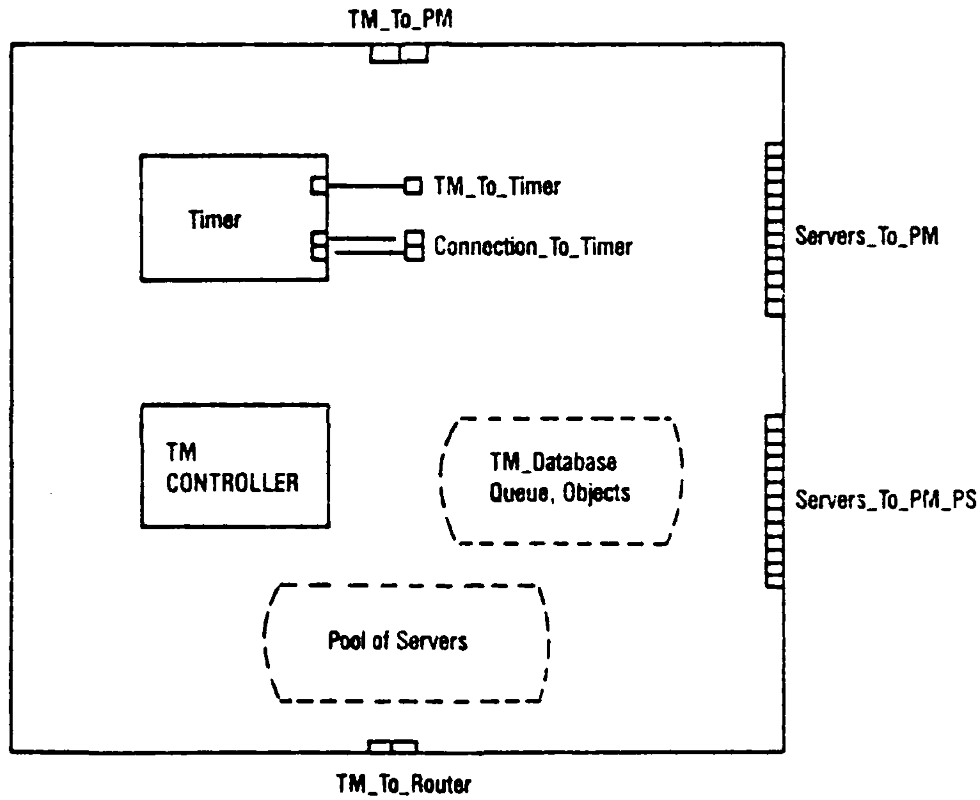
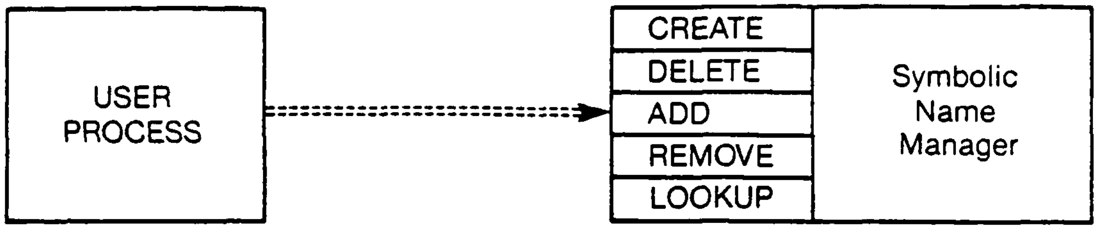


Figure 1-9



CREATE — Create a new context object
DELETE — Delete an existing context object
ADD — Add a new name = > UID mapping
REMOVE — Remove an existing name – > UID mapping
LOOKUP — Return the UID associated with a name

Figure 1-10. User Logical View

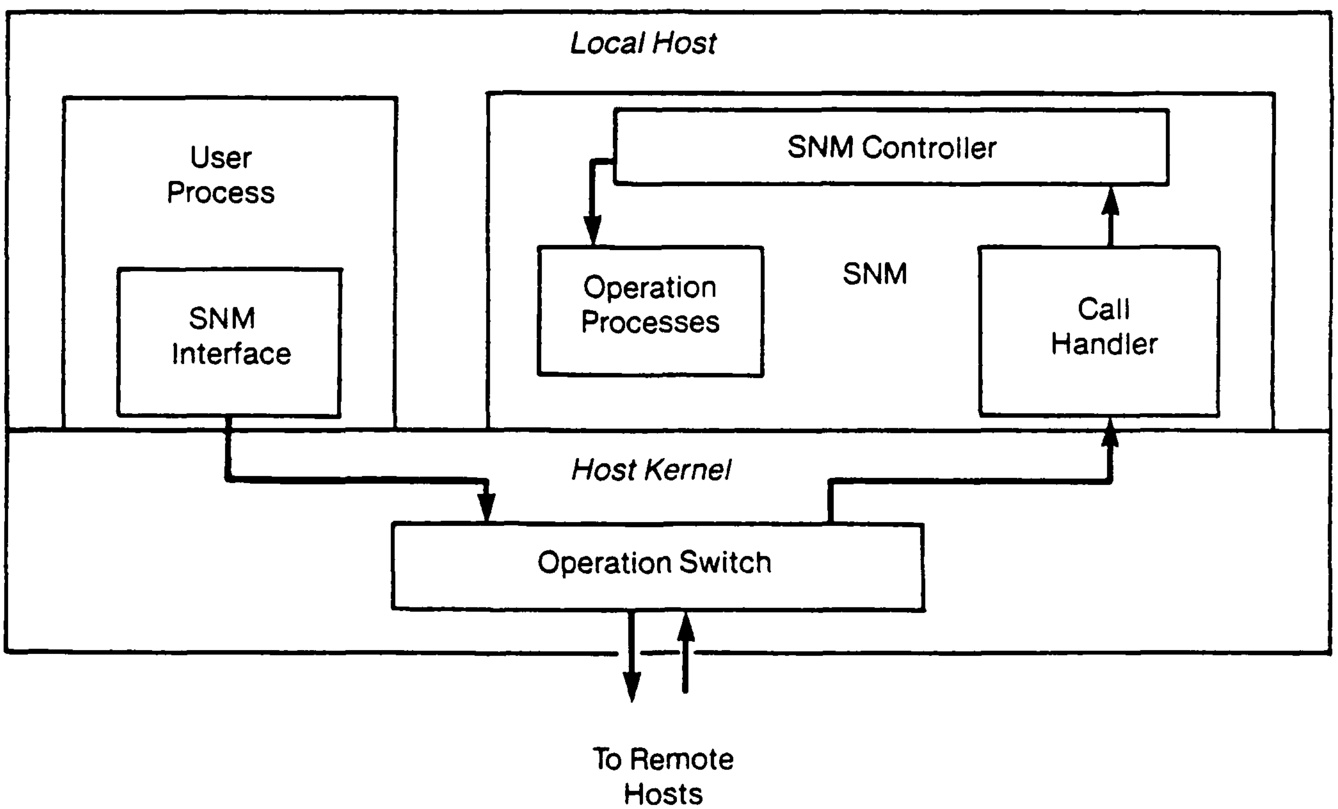


Figure 1-11. SNM Architecture

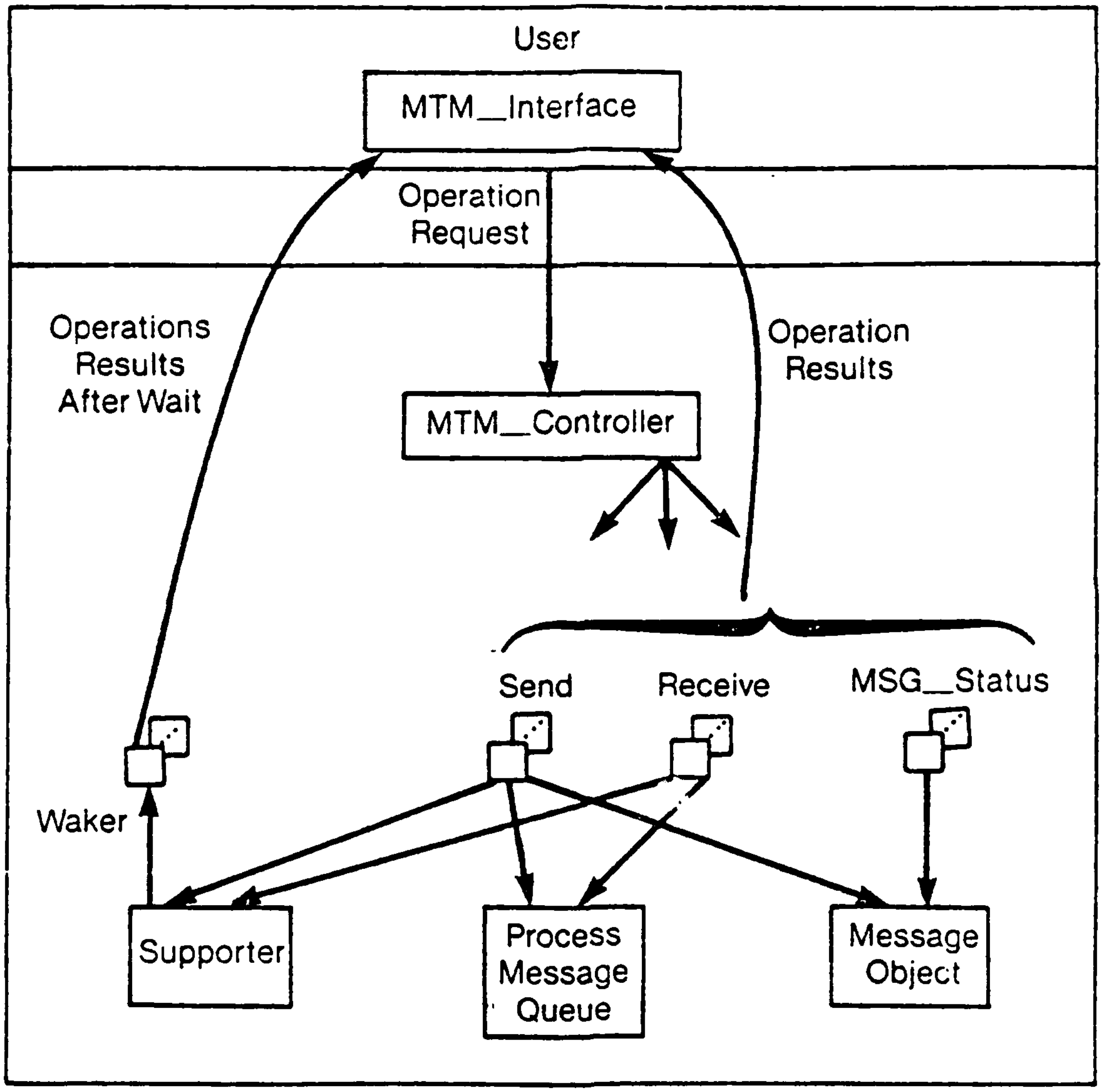


Figure 1-12. Components of Message Management

Chapter 2

KERNEL DESIGN

2.1 INTRODUCTION

This document presents the design of the ZEUS kernel. It describes the mechanisms necessary to implement the kernel interface of section 1.2.1.1 of the guidebook. It also is an elaboration of the system design overview of section 1.3.2 of the guidebook.

The environment consists of a cluster of hosts connected by a CSMA/CD network with reliable broadcast. A host may possess secondary storage.

The kernel is accessed by the type managers on a host. To ensure type transparency, all objects or calls passed as parameters to the kernel are converted to bit strings. It is assumed that the type managers import interface packages that will convert objects into bit strings and vice versa. The kernel has some data types that are accessible to type managers. Chief among these are the unique identifier, the extended unique identifier, and, as mentioned before, the message and object string types.

The goals of the kernel are:

1. Quick execution of kernel procedures.

2. Transparency to permit easy integration of new type managers.

3. Flexibility to permit replacement of storage management strategies and communication protocols.

4. Simplicity to permit easy modeling.

2.1.1 Kernel Interface

This section provides the reader of this document an overview of the kernel. The kernel provides three services that are easily separable for reasons of manageability. To ensure that the reader does not lose sight of the big picture, this document presents each one of these three services individually. This section outlines the entire kernel and the reader is encouraged to turn back to this section as often as is necessary.

Each of the three functions (RPC, storage, and UID generation) have a set of procedures in the kernel interface. These procedures are invoked by the

type managers. The kernel functions have a set of utility packages which they may share with each other. The kernel functions are accomplished by a set of ADA tasks.

The kernel interface describes the visible portion of the kernel and consists of kernel visible functions and data types. The kernel interface is separated into the three sets of procedures, one for each of the kernel functions. The kernel interface is described informally in section 1.2.1.1 of the guidebook. The kernel interface is graphically presented in Figure 2-1. The following pseudo ADA code details the sections in this document where descriptions of the interface may be found.

```
PACKAGE kernel IS

  --RPC functions, section 2.2.1
    PROCEDURE make_call; PROCEDURE make_resp;
    PROCEDURE kill_call; PROCEDURE keep_call;
    PROCEDURE get_call; PROCEDURE get_resp;
    PROCEDURE c_status; PROCEDURE r_status;

  --RPC data types, section 2.2.1
  --visible types
    TYPE host_id; TYPE xtnded_uid;
    TYPE message; TYPE call_status;
    TYPE resp_status; TYPE del_option;
    TYPE kill_status; TYPE uid;
  --private types
    TYPE incrnge; TYPE seqrnge;
    TYPE uid;

  --Object Management functions, section 2.3.1
    PROCEDURE get_obj; PROCEDURE put_obj;
    PROCEDURE del_obj; PROCEDURE stable_get;
    PROCEDURE stable_put; PROCEDURE stabl_del;

  --Object Management data types, section 2.3.1
  --visible types
    TYPE simpl_status; TYPE stabl_status;
    TYPE obj_string;

  --UID generation functions, section 2.4.2
    FUNCTION get_uid; FUNCITON build_xt;
    FUNCTION give_host_hint; FUNCTION change_hint;

  --UID generation data types, section 2.2.1
  --defined were uid, xtnded_uid, host_id.

END PACKAGE kernel
```

2.1.2 The Kernel Structure

The kernel consists of a task structure, some of which is shared by the different kernel functions. The kernel body outline presented here points out the major components and indicates the relevant sections of this document where they can be found. An overview of the kernel structure can be found in section ___ of the guidebook.

```
PACKAGE BODY kernel IS

  --the bodies of the RPC, object management and UID
  --generation procedures.
  --RPC, section 2.2.5
  --Object Management, section 2.3.8
  --UID generation, section 2.4.3.1
  --internal types within the kernel.
  --RPC, section 2.2.7.2; UID generation, section 2.4.3.3
    TYPE pkt-resp; TYPE pkt_class;
    TYPE pkt_info; TYPE ballast;
    TYPE packet; TYPE nt_state;
  --object management, section 2.3.4.1
    TYPE secst_addr; TYPE smp_dir_entry;
    TYPE stb_dir_entry; TYPE d_state;

  --utility packages in the kernel
    PACKAGE type_mgr_map; --sections 2.2.2.1, 2.3.3.1
    PACKAGE message_storage; --section 2.2.2.2
    GENERIC PACKAGE the_buffer; --section 2.2.7.3
    PACKAGE packet_mgr; --section 2.2.7.4.1
    PACKAGE rec_pkt_mgr; --section 2.2.7.4.2
    PACKAGE free_storage; --section 2.3.3.2
    PACKAGE stabl_free; --section 2.3.3.3
    GENERIC PACKAGE the_map; --section 2.3.4.2
    GENERIC PACKAGE the_set; --section 2.4.3.3

  --task types used
  --RPC
    TASK BODY call_handler, --section 2.2.4
      PACKAGE send_call; --section 2.2.3.2
      PACKAGE receive_call; --section 2.2.3.2
    END
    TYPE ch_ptr IS ACCESS call_handler;
  --storage management
  --request handlers, section 2.3.4
    TASK TYPE smp_get; TYPE smp_get_ptr;
    TASK TYPE smp_put; TYPE smp_put_ptr;
    TASK TYPE smp_del; TYPE smp_del_ptr;
    TASK TYPE stb_get; TYPE stb_get_ptr;
    TASK TYPE stb_put; TYPE stb_put_ptr;
    TASK TYPE stb_del; TYPE stb_del_ptr;
```

```
--directory, sections 2.3.6.1, 2.3.6.2, 2.3.6.3;
  TASK TYPE smp_dir; TYPE smd_ptr;
  TASK TYPE sbd_dir; TYPE sbd_ptr;
--device controllers
  TASK TYPE d_ctl; TYPE d_ctl_ptr;

--task definitions
  TASK net_tranceiver; --sections 2.2.7.3, 2.4.3.3
  TASK send_driver; --section 2.2.7.4.1
  TASK receive_driver; --section 2.2.7.4.2
  TASK uid_monitor; --section 2.4.3.2
  TASK uid_generation; --

END PACKAGE BODY kernel;
```

The above definitions are not syntactically correct ADA, but the purpose is to give the reader a central point from which to connect all the parts of the kernel.

Some of the modules above contain other packages that are defined inside them. Typically, these are the task types whose instances need to manage data separately.

The figures 2-3, 2-4 and 2-5 give a graphic overview of RPC, object storage, and uid generation functions, respectively.

2.2 THE REMOTE PROCEDURE CALL STRUCTURE

Object invocations are made from one type manager to another. The source type manager may or may not be on the same host as the destination type manager. The kernel's responsibility is to deliver the call and the response to the call from the source to the destination and vice versa, respectively. This does not preclude the invocation of a type operation at the same type manager on the same host from passing through that host's kernel. Thus, the definition of Remote Procedure Call (RPC) includes the notion of the information of type operations on the same host.

A remote procedure call is accomplished as follows:

1. The source type manager executes a 'make_call' operation at the kernel interface. The kernel returns a unique identifier for the call.

2. The kernels of the source and destination machines use their internal structure to deliver the call to the destination machine kernel.

3. The kernel of the destination machine returns an ack for the call to the kernel on the source machine after the destination type manager issues a 'get_call' operation at the kernel interface.

4. On completion of the type operation, the destination type manager issues a 'make_resp' operation at the kernel interface.
5. The kernels of the source and destination machines use their internal structure to deliver the call to the source machine kernel.

6. The kernel of the source machine returns an ack for the call to the kernel of the destination machine after the source type manager issues a 'get_resp' operation at the kernel interface.

Figure 2-2 is a graphical picture of the messages exchanged for the RPC.

A design constraint imposed on the kernel is that it must know nothing about the structure or operations of the type managers. To achieve this. one must accomplish control independence and data independence. Control independence is achieved by permitting calls to be made from the type managers to the kernel only. This forces the use of four calls so that the kernels and type managers at the source and destinations are blocked for as little time as possible. Of course, this differs from strict procedure call semantics. To achieve data independence, all callers of a type manager import an interface to pack calls to that type manager into bit strings, and to unpack responses from that type manager from bit strings into proper values. Similarly, a type manager has an interface that converts bit strings into type operation calls and response into bit strings.

2.2.1 The Components of the RPC Function

The RPC function interfaces to the type manager via a set of kernel procedures. Within the kernel these procedures make entry calls to tasks called call handlers. At a given kernel there exists one call handler for each object type manager on that host. Each call handler monitors all the outstanding calls from and to its type manager.

The kernel procedures detect local calls and directly pass information to both the source and destination type manager call handlers. In effect the kernel procedures simulate the network. Thus for local calls;

1. The make_call procedure would deliver the call to the source call handler and then deliver the call to the destination call handler.

2. The make_resp procedure would deliver the response to the destination call handler and then to the source call handler.

3. The give_call procedure would get the call from the destination call handler and then give a call ack to the source call handler.

4. The give resp procedure would get the response from the source call handler and then give a response ack to the destination call handler.

For remote calls, the call handlers are polled by the network handler for calls, call acks, responses, and response acks. The network handler also re-assembles calls, call acks, responses, and response acks, and delivers them to the appropriate call handler. The network handler consists of a send driver and receive driver to handle packetizing and reassembly of messages. The actual network interface is a task called the net_tranceiver which handles packets from and to the ethernet controller. This task also sends and receives packets to and from the uid_generation task.

2.2.2 Comparison to Another Effort

Birrell and Nelson have implemented RPC [1] at Xerox PARC. They use a stub in the caller and callee. Thus, a type manager or process would have to import a stub for each type manager it accesses. In addition, the type manager would have a stub to handle calls made to it.

The stubs together comprise the call handler and the send and receive driver in the ZEUS design. The interfaces pass their addresses to each other so that the kernel can directly call them. All this makes for a faster implementation, but it is not as general as ZEUS.

2.2.3 Externally Visible RPC Procedure Calls of the Kernel

The externally visible parts of the kernel that relate to the RPC are declared here. These will include the types visible and the procedures visible to the type manager.

```
PACKAGE kernel IS
  --define uids, extended uids which are used by the type
  --managers to identify objects and remote calls
  TYPE uid IS LIMITED PRIVATE;
  TYPE host_id IS RANGE 0..1023;
  TYPE xtnded_uid IS
    RECORD
      host_hint; host_id;
      typeuid, instanceuid, versionuid; uid;
    END
  --a message is an array of bits
  TYPE message IS ARRAY (RANGE < >) OF BOOLEAN;

  --define the status values that are returned by
  --calls to the kernel. There is a different status
  --type for calls and responses to calls.
  --call status can be
  --not_dlvred - the status of a call that has not
  -- reached the destination type manager
  --dlvred - call reached destination type manager
  --resp_here - call's response has arrived
```

```
--does_not_exist - object addressed does not exist
--rcvr_failed - the destination host is down
--wait_buffer_full - no space locally; hold call
TYPE call_status IS (not_dlvred, dlvred, resp_here,
                     does_not_exist, rcvr_failed,
                     wait_buffer_full);

--resp_status is the status of a response to a call
--not_reached - response not reached source type manager
--reached - response reached source type manager
--accepted - response accepted by source
--sndr_failed - the source host is down
--wait_buffer_full - no space locally; hold response
TYPE resp_status IS (not_reached, reached, accepted,
                     sndr_failed);

--del_option permits a type manager to specify
--damage containment should the source or
--destination host of a call failed.  The options are:
--failure_revoke - the call must be cancelled if
-- the source or destination fails
--failure_continue - the call must continue despite failure
--failure_inform - the caller or callee's type manager
-- must be informed of the failure
TYPE del_option IS (failure_revoke, failure_continue,
                    failure_inform);

--kill_status returns the status of a call
--whose deletion the type has requested
--dead_all_over - call deleted
--not_dead_yet - call still not deleted
--does_not_exist - no call exists
TYPE kill-status IS (dead_all_over, not_dead_yet,
                    doer_not_exist);

--****put in the types for object storage and
--****uid generation over here.

--here start the visible functions of the kernel
--each function is described along with its parameters
--calls to these functions may be delayed because
--the kernel may be busy.

--make_call is invoked in order to make a remote
--procedure call.  Its parameters are:
--typ_uid - uid of the type manager making the call
--source, destination - xtnded uids of the caller and callee
--the_call - a call as a string of bits.  This will
-- contain encoded in it the process
-- xtnded uid, the principal xtnded_uid
-- the call and the parameters of the call
--call_option - the del_option for this call
```

```
--call_uid - the uid assigned to this call
--the_status - the status of the call
PROCEDURE make_call (typ_uid :  IN uid;
    source, destination :  IN extnded_uid;
    the_call :  IN message; call_option :  IN del_option;
    call_uid :  OUT uid; the_status :  OUT call_status);

--make_resp is invoked by the callee in order to
--return a response to a call.  Responses must be
--returned so that a caller knows when the
--callee has terminated.  The parameters are
--type_uid - uid of the type manager making the response
--the_resp - a message that contains in it the response
-- to the call
--resp_option - the del_option for the response
--the_status - the response status

PROCEDURE make_resp (type_uid :  IN uid; the_resp :  IN
    message; source, destination :  IN xtended_uid;
    call_uid :  IN uid, resp_option :  IN del_option;
    the_status :  OUT resp_status);

--kill_call is invoked by the caller or the callee
--to delete a call.  It may be invoked independently
--or when the del_option is failure inform and
--the type manager is informed of a call failure.
--The parameters are:
--call_uid - is the uid of the call
--type_uid - is the uid of the type manager
--the_status - is the kill_status of the call.

PROCEDURE kill_call (call_uid; typ_uid :  in uid;
    the_status :  out kill_status);

--keep_call lets the type manager tell the kernel to
--preserve the call in spite of failure.  The
--parameters are:
--the_uid - call identifier
--typ_uid - type manager uid
--the_status - status of the call

PROCEDURE keep_call (call_uid, type_uid :  IN uid;
    the_status :  OUT kill_status);

--get_call and get_response get calls directed
--to the type manager and responses to calls
--made to the type manager respectively.
--The parameters for get_call are:
--the_uid - call identifier
--type_uid - type manager uid
--the_call - call text
--source - caller's identity
--destination - destination's identity
```

```
  --The parameters for get_resp are:
  --the_uid - call identifier
  --type_uid - type manager uid
  --the_resp - response to call
PROCEDURE get_call (typ_uid :  IN uid, the_uid :  OUT uid;
    the_call :  OUT message; source, destination :  OUT
    xtnded_uid);

PROCEDURE get_resp (type_uid :  IN uid; the_uid :  OUT uid;
    the_resp :  OUT message);

  --c_status and r_status return the status
  --of a call or a response message to the caller
  --and the callee type managers.  The parameters are:
  --the_call - call identifier
  --typ_uid - type manager uid
  --the_status - call_status and resp_status for
  -- c_status and r_status respectively.

PROCEDURE c_status (the_call, typ_uid :  IN uid;
    the_status :  OUT call_status);

PROCEDURE r_status (the_call, typ_uid :  IN uid,
    the_status :  OUT resp_status);

--***here follow the procedure headers for
--***access to object storage and uid
--***generation.

PRIVATE

--here follow the declarations of the limited private
--types uid and xtnded uid.

TYPE incrnge IS RANGE 0..(2**32-1);
TYPE seqrnge IS RANGE 0..(2**22-1);
TYPE uid IS
  RECORD
    origin_host :  host_id;
    incarnation :  incrnge;
    sequence :  seqrnge;
  END RECORD;
--*** here are declared other private types.
END PACKAGE kernel;
```

2.2.4 The Utility Structures Within the Kernel

2.2.4.1 Mapping to Call Handlers

The next part of the RPC design is to describe the means of transfer of information inside the kernel. The kernel, as was mentioned earlier, consists

of one call handler for each type manager and a network handler. The mapping between type manager and call handler is achieved by a package that maps (type) uids to access variables of the call handler task type. This package is called the type_mgr_map.

```
PACKAGE type_mgr_map IS
  function get_chlr (typ_uid :  IN uid) returns
    ch_ptr;

  --This returns a pointer to the call handler task

  PROCEDURE instl_chlr (typ_uid :  IN uid:  th_chlr:  IN
    ch_ptr);

  --sets up an entry in the map for the call
  --handler.  The package insures that only one
  --insertion is occurring at a time.

  PROCEDURE delete_chlr (typ_uid :  IN uid);

  --deletes an entry from the map.  This call
  --must exclude other calls.
  --The package will internally use a task
  --to ensure mutual exclusion between
  --the different calls.

  --*Insert the calls that give access to storage
     directory tasks.

END PACKAGE type_mgr_map;
```

2.2.4.2 The Message Storage Manager

The next set of structures that need to be defined are the tables that store calls and responses to calls as well as the state information related to calls. There are a number of options possible. Ideally, storage of calls should be centralized so that storage devoted to calls and software to manipulate them would be minimized.

This design chooses to do the following:

1. Store the messages in a separate structure.

2. Store call and response to call state information in the call handlers.

3. The message storage structure would provide pointers to the messages that are stored along with the message state information. Externally the pointer is a limited private type, but internally it is an integer subtype. Thus access to a message is rapid.

The visible part of message storage is given below:

```
PACKAGE message_storage IS
  TYPE msg_access IS LIMITED PRIVATE;
  FUNCTION put_msg (the msg :  IN message;
                   the_id :  IN uid) RETURNS
                    msg_access;
  FUNCTION get_msg (the_ptr :  IN msg_access call_uid :
                   IN uid) returns message;
  PROCEDURE del_msg (the_ptr :  IN msg_access call_uid :
                    IN uid);

  --the purpose of the above routines is obvious.
  --the install and delete functions exclude
  --all other functions.

END PACKAGE message_storage;
```

2.2.5 The RPC Protocol

The state information of calls to and from a type manager is presented next. A call consists of a number of stages and it is best to break it up into these stages and analyze each stage for its information requirement. Calls from a type manager are considered first.

1. The call is delivered to the call handler.

2. The network handler sends the message and delivers the callee type manager's acknowledgement to the call handler.

3. The network handler receives a response to the call and delivers it to the call handler.

4. The type manager accepts the response and the call handler acknowledges the response to the network handler and thus the callee's type manager.

Calls to a type manager have the following sequence.

1. The network handler receives a call and passes it to the call handler.

2. When the type manager accepts the call, the call handler sends an acknowledgement to the caller.

3. The type manager delivers a response to the call, to the call handler.

4. The network handler takes the response, delivers it to the caller, and receives an acknowledgement of the response.

2.2.5.1 State Information Required to Support the RPC Protocol

For the outgoing calls the information requirements for each state are as follows:

State 1: Call Delivery

i) Call uid.
ii) Call text (the pointer is stored in the call handler).
iii) Source and destination extended uid.
iv) Status of call (initially not delivered or receiver failed; finally delivered or object does not exist).
v) Delete option specified for the call.

Stage 2: Call in progress

i) Call uid.
ii) Source and destination extended uid.
iii) Status of call (initially delivered, finally either receiver failed or response here).
iv) Delete option specified for the call.

Stage 3: Call completed

i) Call uid.
ii) Response message (pointer).
iii) Source and destination extended uid.
iv) Status of call (stays at response here).
v) Delete option specified for the call.

A status of receive failed or object does not exist, may result in the call being aborted by the caller.

The information requirements for each stage of call handling are as follows:

Stage 1: Call reception

i) Call uid.
ii) Call text (pointer to the text).
iii) Source and destination extended uids.
iv) Status of response (initially as not reached but may change to sndr_failed).
v) Delete option for the call.

Stage 2: Call in progress

i) Call uid.
ii) Source and destination extended uids.

iii) Status of response (not reached, sndr_failed).
iv) Delete option for the call.

Stage 3: Response Dispatch

i) Call uid.
ii) Response text.
iii) Source and destination extended uids.
iv) Status of response (initially not reached or sndrfailed, finally reached).
v) Delete option of response.

2.2.5.2 Packages to Support Call State Information Management

From the above analysis it is clear that two packages are necessary in each call handler: one for calls sent, the other for calls received. Each package will have a set of specialized functions that install calls, delete calls, and update the status of the call.

```
PACKAGE send_call IS

  --install sets up a call, its parameters are
  --1.  Call uid
  --2.  Call text pointer
  --3.  Source and destination extended uid
  --4.  Delete option
  --The status of an installed call is not delivered.

  procedure install (the_uid :  in uid; the_call:  in
      msg_access; source, dest :  in xtnded_uid;
      the_opt :  delete_option):

  --call_dlvred updates the call table for a
  --call that has been accepted by the
  --destination's type manager parameters are:
  --Call uid
  -- A side effect is that the msg access
  --pointer is deleted.

  PROCEDURE call_dlvred (the_uid :  IN uid);

  --call_answered installs a pointer to a
  --response for a call.  It also updates
  --the call status to response_here.
  --Parameters are:
  --1.  Call uid.
  --2.  Response text pointer.

  PROCEDURE call_answered (the_uid :  IN uid;
    the_answer :  IN msg_access);
```

```
--select_next. selects the next call whose size is
--less than or equal to cr_'nd to be sent.  In
--effect this is the dispatcher of calls.
--The parameters are:
--1.  Bound on call size.
--2.  Call uid.
--3.  Call text pointer.
--4.  Call source and destination.
--5.  Call delete option.
--Note:  this procedure will not select a local call.

PROCEDURE select_next (cr_bnd :  IN INTEGER the_uid :
  OUT uid; the_call :  OUT msg_access; source,
  destination :  OUT xtnded_uid; the_opt :  OUT
  del_option);
--host_failed is part of damage containment at
--the kernel level.  It is invoked when a
--host fails.  It processes all the calls
--made to the failed host and deletes
--those calls with a delete option
--equal to failure_revoke.

PROCEDURE host_failed (the_host :  IN host_id);

--remove_call.  Removes a call entry from the
--call table.  This routine is invoked by the
--type manager either to remove a call that
--has been responded to or to remove
--a call whose callee host has failed.
--The parameters are:
--1.  Call uid.
--2.  Call text pointer.
--3.  Response text pointer.

PROCEDURE remove_call (the_uid :  IN uid;
  c_ptr, r_ptr :  OUT msg_access);

--give status returns the status of a call

FUNCTION give_status (the_uid :  IN uid)
  RETURNS call_status;

--give_response returns response to some call
--it selects responses by some algorithm.

PROCEDURE give_response (the_uid :  OUT uid; the_call :
  OUT msg_access);

--the package internally requires no
--mutual exclusion on invocations since
--all its routines are invoked in the
--call handler.
```

```
  --give_ack returns the uids of a response that
  --has been received

  PROCEDURE give_ack (the_uid :  OUT uid; the_clr,
    the_cle :  OUT xtnded_uid);

END PACKAGE send_call;
```

The second package handles calls received. It has similar functions to the package send_call. The response status in this package for a call will be not_reached until the response is issued and delivered to the caller's type manager.

```
PACKAGE receive_call IS

  --put_call installs a call in the table.


  PROCEDURE put_call (the_uid :  IN uid; the_call :  IN
    msg_access; source, destination:  in xtnded_uid)

  --call_dlvred removes the call from the table.
  --The uid, source and destination are left in and
  --the status become dlvred.

  PROCEDURE call_dlvred (the_uid :  OUT uid; the_call;
    OUT msg_access; source, destination :  OUT
    xtnded_uid);

  --response_given puts the response text pointer
  --into the table.

  FUNCTION response_given (the_uid :  IN uid;
    the_answer :  IN msg_access; resp_option :
    IN del_option) RETURNS resp_status;

  --select_next releases a remote response provided it
  --is less than the value of cr_bnd.

  PROCEDURE salect_next (cr_bnd :  IN integer; the_uid :
    OUT uid; the_txt :  OUT msg_access;
    source, destination :  out xtnded_uid;
    the_opt :  OUT del_option);

  --host_failed has the same function as in
  --package send_call.

  PROCEDURE host_failed (the_host :  IN hostid);

  --response_acked changes the status of the
  --call entry to accepted.
```

```
        PROCEDURE response_acked (the_call :  IN uid);

        --response_state returns the status of the response.

        FUNCTION response_state (the_call :  IN uid)
          RETURNS resp_status;

        --selects next remote call to be acknowledged.

        PROCEDURE give_call_ack (the_call :  OUT uid;
          the_clr, the_cle :  OUT xtnded_uid);

      END PACKAGE receive_call;
```

2.2.6 The Call Handler

The call handler is a task type that is declared within the kernel. Each call handler contains an instance of send_call and receive_call. An instance of this type is generated for each type manager. Some of the entry points of this task are accessed by the visible procedures of the kernel, others are invoked by the network handler. Each of the entry points of the call handler, when invoked, will call some procedures in the call handler's send_call and receive_call packages to manipulate the state information about the call. This interaction is summarized in Figure 2-3. In addition, Figures 2-4 and 2-5 present this information in more detail.

The entry points in the call handler are similar in function to the entry points of the kernel. The exceptions are the entry calls for the network handler which give and receive calls.

```
      TYPE ch_ptr IS ACCESS call_handler;
      TASK TYPE call_handler is
        ENTRY mc (c_uid :  IN uid; c_data :  IN msg_access;
            c_from, c_to :  IN xtnded_uid;
            c_opt :  IN del_option; c_stat :  OUT call_status);

        --mc receives an entry for the send_call
        --table of this call handler.

        ENTRY mr (c_uid :  IN uid; v_data :  IN msg_access;
          c_from, c_to :  IN xtnded_uid;
          r_opt :  IN del_option; r_stat :  OUT resp_status);

        --mr enters as response to a call.
        --it uses the receive call package of
        --the call handler.

        ENTRY klc (c_uid :  IN uid; c_stat :  OUT
          kill_status; c_ptr, r_ptr :  OUT msg_access);

        --klc kills a call issued by this type manager.

        ENTRY kpc (c_uid :  IN uid; c_stat :  OUT kill_status);
```

```
--kpc instructs the call handler to
--keep a call despite failure at the
--receiver host.

ENTRY rs (c_uid :  IN uid; c_stat :  OUT resp_status);

--response status.

ENTRY cs (c_uid :  IN uid; c_stat :  OUT call_status);

--call_status
ENTRY tr (c_uid :  OUT uid; r_text :  OUT msg_access);

--returns a response to calls to the type mgr.

ENTRY tc (c_uid :  OUT c_text :  OUT msg_access;
  c_from, c_to :  OUT xtnded_uid);

--returns a call to the type manager

ENTRY gtcr (c_uid :  IN uid; cr_text :  IN msg_access;
  c_from, c_to :  IN xtnded_uid);

--gives a call or response to this call handler.

ENTRY gtack (cr_ack :  IN uid; c_from, c_to :
  IN xtnded_uid);

--gives an acknowledgement for a call or
--response message from this call handler.

ENTRY lc (c_uid :  IN uid; cr_text :  IN msg_access;
          c_form, c_to :  IN xtnded_uid);

--local call to destination call handler.

ENTRY lr (c_uid :  IN uid; cr_text :  IN msg_access;
          c_from, c_to :  IN xtnded_uid);

--local response to source call handler.

ENTRY lcack (c_uid :  IN uid; c_from, c_to :  IN xtnded_uid);

--local call ack to source call handler.

ENTRY lrack (c_uid :  IN uid; c_from, c_to :  IN xtnded_uid);

--local response ack to destination call handler.

ENTRY gvcr (cr_bnd :  IN integer; OUT uid; cr_text :
  OUT msg_access; cr_from, cr_to :  OUT xtnded_uid;
  c_opt :  OUT del_option);
```

```
  --gives a call or a response to the
  --network handler.

  ENTRY gvack (cr_ack :  OUT uid; c_from, c_to :  OUT
    xtnded_uid);

  --returns an acknowledgement to a host.

  ENTRY host_down (the_host :  IN host_id);

  --tells of a failed host.  used for damage
  --containment.

END TASK call_handler;

TASK BODY call_handler IS

--count of calls and responses to be delivered
--to the type manager.

ready_calls, ready_resp :  INTEGER := 0;

--calls and responses to be acked.

not_acked_calls, not_acked_resp :  INTEGER := 0;

--count of calls and responses to be sent out

out_calls, out_resp;
out_who :  boolean := true;
failed_id :  host_id := 0;
BEGIN
  LOOP --endless loop
    SELECT
      --calls made by this type manager
      ACCEPT mc(c_uid, c_data, c_from, c_to, copt, c_stat) DO
        send_call.install (c_uid, c_data, c_from,
          c_to, c_opt, c_stat)
        out_calls :  out_calls + 1;
      END;
    OR
      --accept responses to calls.
      ACCEPT mr (c_uid, r_data, c_from, c_to, r_opt,
          r_stat) DO
        r_stat := receive_call.response_given
                    (c_uid, r_data, r_opt);
        IF r_stat := not_reached THEN
          out_resp := out_resp + 1;
        ENDIF
      END
    OR
      --accept kill call entry.
      ACCEPT klc (c_uid, c_stat, c_ptr, r_ptr) do
```

```
      send_call.remove_call (c_uid, c_ptr, r_ptr);
      IF (call_ptr = 0 AND resp_ptr = 0) THEN
      --message deleted but no response.
        c_stat := not_dead_yet.
      ELSE c_stat := dead_all_over;
      ENDIF
    END
  OR
    --accept keep call entry.
    ACCEPT kpc (c_uid, c_stat) DO
      IF send_call.give_status (c_uid) /= does_not_exist
        THEN c_stat := not_dead_yet
        ELSE c_stat := does_not_exist
      ENDIF
    END
  OR
    --response status
    ACCEPT rs (c_uid, c_stat) DO
      c_stat := receive_call.response_state(c_uid);
    END
  OR
    --call status
    ACCEPT cs (c_uid, c_stat) DO
      c_stat := send_call.give_status (c_uid);
    END
  OR
    --accept requests for responses from type mgr.
    WHEN ready resp >0 ==>
      ACCEPT tr (c_uid, r_text) DO
        send_call.give_response (c_uid, r_text);
        ready_resp := ready_resp - 1;
        not_acked_resp := not_acked_resp + 1;
      END
      send_call.remove_call (c_uid, r_text);
    END
  OR
    --accept request for calls from type mgr
      WHEN ready_calls >0 ==>
        ACCEPT tc (c_uid, cr_text, c_from, c_to) DO
          receive_call.call_dlvred (c_uid, cr_text,
            c_from, c_to);
          ready_call := ready_call - 1;
          not_acked_call := not_acked_call + 1;
        end;
  OR
    --accept a new call or response from network handler.
    ACCEPT gtcr (c_uid, cr_text, c_from, c_to) DO
      IF c_uid. origin_host = this_host THEN
        --this is a response
        send_call.call_answered (c_uid, cr_text);
        ready_resp := ready_resp + 1;
      ELSE
```

```
        --this is a call.
        receive_call.put_call (c_uid, cr_text, c_from,
                                c_to);
        ready_calls := ready_calls + 1;
      ENDIF;
    END;
  OR
    --accept an acknowledgement.
    ACCEPT gtack (cr_ack, c_from, c_to) do
      IF c_ack.origin_host = this_host THEN
        --this is a call acknowledgement.
        send_call.call_dlvred (cr_ack);
      ELSE
        --this is a response acknowledgement
        receive_call.response_acked (cr_ack);
      ENDIF;
    END;
  OR
    --accept a local call.
    ACCEPT lc (c_uid, cr_text, c_from, c_to) DO
      receive_call.put_call (c_uid, cr_text, c_from, c_to);
      ready_calls := ready_calls + 1;
    END;
  OR
    --accept a local response.
    ACCEPT lr (c_uid,cr_text, c_from, c_to) DO
      send_call.call_answered (c_uid, cr_text);

      ready_resp := ready_resp + 1;
    END
  OR
    --accept a local call acknowledgement.
    ACCEPT lcack (c_uid, c_from, c_to) DO
      receive_call_dlvred (c_uid);
    END
  OR
    --accept a local response acknowledgement.
    ACCEPT lrack (c_uid, c_from, c_to) DO
      receive_call_response_acked (c_uid);
    END
  OR
    --accept a local response acknowledgement.
    ACCEPT lrack (c_uid, c_from, c_to) DO
      receive_call.response_acked (c_uid);
    END;
  OR
    --accept a request to give out a call or response.
    --out_who decides which of calls or responses it's
    --turn it is.
    WHEN out_calls + out_resp >0 ==>
      ACCEPT gvcr (cv_bnd, c_uid, cr_text, cr_from, cr_to,
          c_opt) DO
        IF (out_who AND out_calls > 0) OR
```

```
            (NOT out_who AND out_resp = 0) THEN
            send_call.select_next (cr_bnd, c_uid, cr_text,
              cr_from, cr_to, c_opt);
            out_calls := out_calls - 1; out_who := FALSE;
          ELSEIF (NOT out_who AND out_resp > 0) + OR
            (out_who AND out_call = 0) THEN
            receive_call.select_next (cr.bnd, c_uid, cr_text,
              cr_from, cr_to, c_opt);
            out_resp := out_resp - 1; out_who := TRUE;
          ENDIF;
        END;
      OR
        --accept a request to give out an ack.
        WHEN (not_acked_calls + not_acked_resp > 0) ==>
          ACCEPT gvack (cr_ack, c_from, c_to) DO
            IF not_acked_resp >0 THEN
              send_call.give_ack (cr_ack, c_from, c_to)
              not_acked_resp := not_acked_resp - 1;
              ELSE
              receive_call.give_call_ack (cr_ack, c_from,
                c_to)
              not_acked_call := not_acked_call - 1;
            ENDIF
          END;
      OR
        --host failure
        ACCEPT host_down (the_host) DO
          failed_id := the_host;
        END;
      END select;
      IF failed_id /= 0 THEN
        send_call.host_failed (failed_id),
        receive_call.host_failed (failed_id);
      ENDIF
      failed_id := 0
    END LOOP;
  END; TASK call_handler;
```

The call_handler task is the end_to_end protocol handler for the ZEUS system. Other than gtcr, gtack, gvcr, gvack, and host_down, the call handler entry points are accessed by the kernel interface functions described earlier. In addition to the entry calls, these functions access some of the other packages. It must be mentioned here that more than one activation of a kernel function can exist simultaneously.

2.2.7 The Kernel Functions for the RPC

Code outlines for each kernel function is presented next after which the type manager interface tasks to receive calls and responses are presented. Following that the network handler design and implementation is presented.

```
PROCEDURE make_call (typ_uid :  IN uid; source, destination :
    IN xtnded_uid; the_call :  IN message;
    call_option :  IN del_option; call_uid :  OUT uid;
    the_status :  OUT call_status);
  the_hndlr :  ch_ptr, --call handler pointer
  the_ptr :  msg_access; --message pointer
begin
--get uid for the call
  call_uid := get_uid;
--put message into message database
--first check that destination is up.
  IF rec_pack.host_up (destination.host_hint) THEN
    the_status := NOT dlvred;
    the_ptr := message_storage.put_msg (the_call,
      call_uid);
    --put the message into the call_handler.
    IF the_ptr > 0 THEN
        the_hndler := type_mgr_map.get_chlr (typ_uid);
        the_hndler.mc (call_uid, the_ptr, source,
          destination, call_option, the_status);
      ELSE the_status := wait_buffer_full;
      ENDIF
    --check for local call
    IF source.host_hint = destination.host_hint THEN
      the_hndler := type_mgr_map.get_chlr (destination.typeuid);
      the_hndlr.lc (call_uid, the_ptr, source, destination);
    ENDIF
    ELSE
      the_status := rcur_failed;
    ENDIF;
END make_call;
```

The above routine uses a procedure host_up and a function get_seq. The former is part of a package that the kernel uses to

detect host failures while the latter is part of a package the kernel uses to get sequence numbers. Both these packages are specified later.

```
procedure make_resp (typ_uid :  IN uid; the_resp :  IN
  message; source, destination :  IN xtnded_uid;
  call_uid :  IN uid; resp_option :  IN del_option;
  the_status :  OUT resp_status) IS
  the_ch :  ch_ptr; --call handler
  the_ptr :  msg_access; --message pointer.
BEGIN
  --put response in database.
  IF rec_pack.host_up (source.host_hint) THEN
    the_status := not_reached;
    the_ptr := message_storage.put_msg (the_resp,
      call_uid);
      IF the_ptr > 0 THEN
        --insert response into call_handler
```

```
          the_ch := type_mgr_map.get_chlr (typ_uid);
          the_ch.my (call_uid, the_ptr, source,
            destination, resp_option, the_status);
          --check for local response
          IF source.host_hint = destination.host_hint THEN
            the_ch := type_mgr_map.get_chlr (source.type_uid);
            the_ch.lr (call_uid, the_ptr, source, destination);
          ENDIF
        ELSE the_status := wait_buffer_full;
        ENDIF;
  ELSE
    the_status := sndr_failed;
  ENDIF
END make_resp;
```

The make_resp and make_call routines are very similar and use similar packages. The mapping of type uid to call handler may be bypassed by making the call handler points a limited private type visible from the kernel.

```
PROCEDURE kill_call (call_uid, typ_uid :  IN uid;
    the_status :  OUT kill_status) is
  the_ch :  ch_ptr;
  in_msg, out_msg :  msg_access;
BEGIN
  --get call handler and kill call
  the_ch := type_mgr_map.get_chlr (typ_uid);
  the_ch.klc (call_uid, the_status, in_msg, out_msg);
  --delete messages from message storage
  IF in_msg /= 0 THEN
    message_storage.del_msg (in_msg, call_uid);
  ENDIF;
  IF out_msg /= 0 THEN
    message_storage.del_msg (out_msg, call_uid);

  ENDIF;
END kill_call;
```

In the above procedure it can be ensured that only one message is deleted by removing a call or a response once it is accepted by the type manager.

```
PROCEDURE keep_call (call_uid, typ_uid :  IN uid;
  the_status :  OUT kill_status);
  the_ch :  ch_ptr;
BEGIN
  --get call handler
  the_ch := type_mgr_map.get_chlr (typ_uid);
  the_ch.kpc (call_uid, the_status);
END keep_call;
```

The next two kernel interface routines get_call and get_resp are similar. They remove messages directed to the type manager.

```
PROCEDURE get_call (typ_uid :  IN uid; the_uid :  OUT uid;
  the_call :  OUT message; source, destination :  OUT
    xtnded_uid) is
  the_ch :  ch_ptr;
  in_msg :  msg_access;
BEGIN
  the_ch := type_mgr_map.get_chlr (typ_uid);
  the_ch.tc (the_uid, in_msg, source, destination);
  --get the message.
  the_call := message_storage.get_msg (in_msg, the_uid);
  --check if local call.
  IF source.host_hint = destination.host_hint THEN
    the_ch := typ_mgr_map.get_chlr (source.typeuid);
    the_ch.lcack (the_uid, source, destination);
  ENDIF
  --delete the message.
  message_storage.del_msg (in_msg, the_uid);
END get_call;

PROCEDURE get_resp (typ_uid :  IN uid; the_uid :  OUT uid;
  the_resp :  OUT message) is
  the_ch :  ch_ptr;
  out_msg :  msg_access;
BEGIN
  --get call handler.
  the_ch := type_mgr_map.get_chlr (typ_uid);
  the_ch.tr (the_uid, out_msg);
  --get the response and delete the message.
  the_resp := message_storage.get_msg (out_msg, the_uid);
  message_storage.del_msg (out_msg, the_uid);
  --check if local call.
  IF source.host_hint = destination.host_hint THEN
    the_ch := type_mgr_map.get_chlr (destination.typeuid);
    the_ch.lrack (the_uid, source, destination);
  ENDIF
END get_resp;
```

The procedure bodies of c_status and r_status are similar because they query the call handler about the call or response status.

```
PROCEDURE c_status (the_call, typ_uid :  IN uid;
  the_status :  OUT call_status) IS
  the_ch :  ch_ptr ; --call handler
BEGIN
  --get call handler.
    the_ch := typ_mgr_map.get_chlr (typ_uid);
    the_ch.cs (the_call, the_status);
END c_status;

PROCEDURE r_status (the_call, typ_uid :  IN uid;
  the_status :  OUT resp_status) IS
  the_ch :  ch_ptr; --call handler;
BEGIN
  --get call handler
```

```
    the_ch := typ_mgr_map.get_chlr (typ_uid);
    the_ch.rs (the_call, the_status);
  END r_status.
```

2.2.8 Tasks Within a Type Manager to Receive Calls and Responses Asynchronously

As mentioned earlier, the type managers should set up tasks so that they can receive calls and responses in an asynchronous manner instead of waiting for them. These tasks are declared inside the type manager, and will deliver calls and responses to the main task of the type manager.

The get call task, for instance, performs the following loop. It first makes a kernel get_call invocation. When the task receives a call it makes an entry call to the type manager. It waits until the type manager accepts the new call. When the call has been accepted the get_call task repeats the above sequence. The get_response task obtains responses to calls in a similar manner.

```
TASK gc_tsk IS
  --no entries
TASK BODY gc_tsk IS
  the_typ, the_uid :  uid; the_call :  message;
  svc, dest :  xtnded_uid;
BEGIN
  the_typ :- <type uid>;
  LOOP
    kernel.get_call (the_type, the_uid, the_call, src,
      dest);
    tm.put_call (the_uid, the_call, src, dest);
  END LOOP;
END gc_tsk;
```

2.2.9 The Network Handler

The networ'. handler sends and receives calls and packets for the kernel. Its functions are to:

1. Poll each handler for calls, responses, or acknowledgements to be sent. On receiving one of these, the network handler packetizes it and sends it off packet by packet over the ethernet controller. For simplicity we assume that one message is being sent at any given time. Timeouts are used to detect failures of packet transmission.

2. Receive and assemble packets into calls and responses to calls. Once a call or response is re_assembled the message is delivered to the proper type manager.

3. Receive and send packets that are signals to start distributed uid incarnation field generation and host recovery computations.

2.2.9.1 The Network Handler Architecture

The architecture of the network handler is a collection of tasks each of which has a different function. The basic task is one which receives and transmits packets. At any time, it interacts solely with one of the following sets of tasks.

1) Send driver for packets.

2) Receive driver for packets.

3) UID generation task.

The packet tranceiving task has two interrupts from the ethernet controller which are rdy_to_snd and rdy_to_rec. The former takes a packet to send on the ether, while the latter delivers a packet that arrived on the ether. The packet tranceiving task waits on the entry points of these two interrupts as well as on its entry points for the sequence generation and recovery protocol tasks that handle outward going packets.

A rendezvous at the rdy_to_rec entry point will result in a rendezvous at the entry point in the packet tranceiving task that accept entry calls to transfer packets to the receive task.

Sequence number generation, restart, and roll call protocols are described in the sequence generation section.

A number of figures describe the interactions among the different network handler components. Figure 2-6 describes the sequence of events to send messages and packets, while Figure 2-7 describes the sequence of events to receive messages and packets. Figure 2-8 allows quick reference to the net_tranceiver tasks' entry points. Finally, Figure 2-9 is a timing diagram of how packets are delivered over the network. The reader should refer to these figures often in order to understand the ADA code.

2.2.9.2 Data Handled by the Network Handler

The first step here is to describe the types of packets that will be sent and received over the network. These will include data packets (dtp), acknowledgements for data packets (adtp), new incarnation packet (nip), host restart packet (hrp), first host answer packet (fhap), start roll call packet (srcp), and response roll call packet (rrcp).

```
TYPE pkt_resp IS (no_ack, ack, badata, nobuff, toobig);

TYPE pkt_class IS (dtp, adtp, nip, nir, hrp, fhap, srcp, rrcp);
--dtp data packet
--adtp data acknowledgement packet
--nip new incarnation packet
--nir new incarnation response
--hrp host restart packet
--fhap first host answer packet
--srcp start roll call packet
--rrcp response roll call packet

TYPE pkt_info IS
  RECORD
    the_id :  uid;
    the_pos, the_tot :  integer;
    the_resp:  pkt_resp;
  END

TYPE ballast IS ARRAY (<range>) OF BOOLEAN;

TYPE packet (class :  pkt_class) IS
```

2.2.9.3 Data Handled by the Network Handler

The first step here is to describe the types of packets that will be sent and received over the network. These will include data packets (dtp), acknowledgements for data packets (adtp), new incarnation packet (nip), host restart packet (hrp), first host answer packet (fhap), start roll call packet (srcp), and response roll call packet (rrcp).

```
TYPE pkt_resp IS (no_ack, ack, badata, nobuff, toobig);

TYPE pkt_class IS (dtp, adtp, nip, nir, hrp, fhap, srcp, rrcp);
--dtp data packet
--adtp data acknowledgement packet
--nip new incarnation packet
--nir new incarnation response
--hrp host restart packet
--fhap first host answer packet
--srcp start roll call packet
--rrcp response roll call packet

TYPE pkt_info IS
  RECORD
    the_id :  uid;
    the_pos, the_tot :  integer;
    the_resp:  pkt_resp;
  END

TYPE ballast IS ARRAY (<range>) OF BOOLEAN;
```

```
TYPE packet (class :  pkt_class) IS
RECORD
  host_addr :  RANGE 0..2047;
  --host address can either be from 1 - to 1023
  --for point to point or broadcast communication or
  --from 1024 to 2047 for multipoint communication.

  CASE class IS
    WHEN dtp =>
      pkt_info, --packet data
      aknk_info :  pkt_info; --piggback ack/nack
      pkt_data :  ballast;
    WHEN adtp =>
      aknk_info :  pkt_info;
      ack_packing :  ballast;
      --null
      --put in package definitions for incarnation request,
      --roll calland host restart computations.


  END CASE
END RECORD packet;
```

2.2.10 The Net Tranceiver Task

Having described the data that will be transferred in the network handler, the next step is to define the interfaces of the tasks that comprise the network handler. The packet tranceiver task is described first. As mentioned before, this task has two entry points that represent ethernet controller interrupts. In addition to these, it has entry points to give or receive packets to the other tasks in the network handler. Each of these is specified below.

```
TASK net_tranceiver IS

  ENTRY rdy_to_snd;
    FOR rdy_to_snd USE AT 8#...#;
  --entry for sending packets
  ENTRY rdy_to_rec;
    FOR rdy_to_rec USE AT 8#...#;
  --entry for receiving packets.
  ENTRY snd_data (the_pkt :  IN packet);
  --entry to send data and data acknowledge packets
  ENTRY get_data (the_pkt :  OUT packet);
  --entry to receive data and data acknowledge packets
  ENTRY put_acks (the_aknk :  IN pkt_info);
  --acks/naks from receiver task
  ENTRY get_ack (the_aknk :  OUT pkt_info);
  --pkt info of packets acked
  ENTRY give_ack (the_aknk :  OUT pkt_info);
  --pkt info of packet received (packets to be acked)

  --entries for uid generation in section 4.3.3
```

END TASK net_tranceiver;

The tranceiver controls traffic to and from a host. Under normal operation the tranceiver will receive and send data packets and data acknowledgement packets. However, the receiving of a host restart, new incarnation, or start roll call packets will result in the tranceiver entering a special mode to cater especially to the incarnation request, restart, or roll call computations. In this mode, the tranceiver will refuse to send or receive packets other than those for the incarnation request, restart, or roll call computation. This will ensure that timing constraints can be imposed on these computations.

Buffering of packets and acknowledgements is also of a concern here. Buffers are needed for incoming and outgoing packets and incoming and outgoing acks or naks. The handling of packets or the sending side is as follows:

1. The net_tranceiver receives a packet from the send_driver task and places it, if possible, in the data_out buffer.

2. The net_tranceiver, when it receives a request to send a packet from the ethernet controller, will dispatch the packet.

3. Packet information from the acks_in buffer is delivered to the send_driver task. This allows packets that were sent earlier to be acknowledged.

The handling of packets on the receiver side is more complex.

1. The net_tranceiver receives a data packet from the ethernet controller. If there is space in the data in buffer it places it there. If there is no space it rejects the packet and places a nack for the packet in the acks_out buffer. If this buffer has no space then the packet can be deemed lost.

2. Packets in the data_in buffer are given to the receive_driver task when it requests them.

3. The receive_driver returns acks or nacks to the net_tranceiver if there is space in the acks_out buffer (condition for the rendezvous).

4. The net_tranceiver then gives acks/nacks from the buffer acks_out to the send_driver. The send_driver then sends these acks/nacks either as separate packets or as piggybacked acks/nacks depending on the line control protocol.

The handling of acks/nacks received by the net_tranceiver is as follows:

1. Acks or nacks are received either separately or piggybacked on a data packet. If there is space in the acks_in buffer they are put into that buffer. Otherwise, they are lost.

2. When the send_driver requests for acks/nacks of packets sent from this host they are delivered to it from the acks_in buffer.

With all these buffers, specifying them explicitly in the net_tranceiver task would be cumbersome. To alleviate this, a generic package called the buffer is declared with the buffer size and buffer element as parameters. Each buffer then is an instantiation of this generic package, thus permitting a simple specification of the system.

```
GENERIC
  size :  POSITIVE
  TYPE elem IS PRIVATE;
PACKAGE the_buffer IS
  FUNCTION is_full RETURNS BOOLEAN;
  FUNCTION is_empty RETURNS BOOLEAN;
  PROCEDURE add_elem (e :  IN elem);
  FUNCTION get_elem RETURNS elem;
  --add_elem and get_elem must be called only
  --after NOT is_empty and NOT is_full return
  --true
END PACKAGE the_buffer;
```

This generic package will allow the various buffers in the system to be specified and used easily.

```
TASK BODY net_tranceiver IS
  out_dt_size :  INTEGER CONSTANT :=
  in_dat_size :  INTEGER CONSTANT :=
  out_ack_size :  INTEGER CONSTANT :=
  int_ack_size :  INTEGER CONSTANT :=
  --the buffer sizes for outgoing data, incoming data,
  --outgoing acks/naks and incoming acks/naks
  --respectively.
  data_out IS NEW the_buffer (out_dt_size, packet);
  data_in IS NEW the_buffer (in_data_size, packet);
  acks_out IS NEW the_buffer (out_ack_size, pkt_info);
  acks_in IS NEW the_buffer (int_ack_size, pkt_info);
  --the buffers are instantiated.
  out_hard, in_hard :  packet;
  FOR out_hard USE AT 8#...#;
  FOR in_hard USE AT 8#...#;
  --hardware buffers.
  TYPE nt_state IS (rpc_alg, uid_alg, rlc_alg, hrp_alg,
    start_up);
```

```
  --state of the net_tranceiver task
  the_state:  nt_state := start_up;
  the_hdr; the_aknk :  pkt_info;
  the_buf :  packet;
  --temporary storage.
BEGIN
  --put in code for start-up state here
  --specified in section 2.4.3.3
  LOOP --infinite loop
  WHILE the_state = rpc_alg LOOP
  --normal operation
  SELECT
    WHEN (NOT data_out.is_empty OR (E'snd_seq > 0)) ==>
    --prefer uid_generation packets over RPC packets
      ACCEPT rdy_to_snd DO
        SELECT
          ACCEPT snd_seq (the_pkt :  IN packet);
            out_hard := the_pkt;
          END
        ELSE
          out_hard := data_out.get_elem;
        END select;
      END
  OR
    ACCEPT ready_to_rec DO
      the.buf := in_hard;
    END
    CASE the_buf.class IS
    --handle data packets
    WHEN dtp =>
      the_hdr := the_buf.pkt_info;
      the_aknk := the_buf_aknk_info;
      IF data_in.is_full then
        the_hdr.the_resp := no_buff;
        IF NOT acks_out.is_full THEN
          acks_out.add_elem (the_hdr);
        ENDIF;
        --send nak for buffer full
      ELSE
        data_in.add_elem (the_buf);
      ENDIF;
      IF the_aknk.the_resp /= no_ack THEN
        IF NOT acks_in.is_full THEN
          acks_in.add_elem (the_aknk);
        ENDIF
        --put ack away
      ENDIF;
    WHEN adtp =>
      --handle acks
      IF NOT acks_in.is_full THEN
        acks_in.add_elem (the_buf.aknk_info);
      ENDIF;
```

```
        --put in handling for the other packet types from
        --section 2.4.3.3.
      END case;
    OR
      WHEN NOT data_out.is_full =>
        ACCEPT snd_data (the_pkt) DO
          data_out.add_elem (the_pkt)
        END
    OR
      WHEN NOT (data_in.is_empty) =>
        ACCEPT get_data (the_pkt) DO
          data_in.get_elem (the_pkt);
        END;
    OR
      WHEN NOT acks_out.is_full =>
        ACCEPT put_acks (the_aknk) DO
          acks_out.add_elem (the_aknk);
        END;
    OR
      WHEN NOT acks_in.is_empty =>
        ACCEPT get_ack (the_aknk) DO
          the_aknk := acks_in.get_elem;
        END;
    OR
      WHEN NOT acks_out.is_empty =>
        ACCEPT give_ack (the_aknk) DO
          the_aknk := acks_out.get_elem;
        END;
      END select;
    END loop;
      --the specification of the handling of packets to
      --generate incarnation numbers, roll call protocols
      --and host restart is given in section 2.4.3.
    END loop;
    END net_tranceiver;
```

2.2.10.1 The Network Receive and Send Tasks

The network receive and send tasks are together equivalent to the transport and line control protocol of a traditional computer network. They interface with the call handler tasks on the one hand and the net_tranceiver task on the other.

On the call handler side these tasks deal with two types of data entities: messages (calls, responses) and message acks (call acks, response acks). The send task breaks these data entities into packets and gives them to the net_tranceiver. The receive task obtains packets from the net_tranceiver, re-assembles them into messages or message acks, and then delivers them to the appropriate call handler.

The send and receive tasks handle the send and receive functions of a line control protocol, too. The architecture of the send and receive tasks is of two tasks, the send driver and the receive driver, along with two packages

to manage packets that have been sent and received. These abstractions handle all the line control, packetization, and re-assembly details.

A possible simpler design would be to have a kernel have a single call being broadcasted at any time. This will permit upper bounds to be placed on the storage requirements of calls being sent or received.

2.2.10.1.1 The Send Task

The send task is divided into two parts. The first is a packet manager that handles the packetization of messages (calls, responses to calls) and of acknowledgements for those messages. The packet manager buffers a few calls at a time and this is done by letting the send task only obtain calls that will fit into the currently available buffer space. The send driver is the task that feeds the packet manager with messages, message acknowledgements, and packet acknowledgements.

```
PACKAGE packet_mgr IS
  TYPE msg_type IS (data, msg_ack);
  --allows send_driver to force the message size to be
  --less than some upper bound.
  PROCEDURE how_many_free (numpkts, numbits :  out integer);
  --returns how many packet blocks and
  --message bits (data) are available.
  PROCEDURE how_many_used (num_pkts, numbits :  out integer);
  --returns how many packet blocks and message
  --bits (data) are used.
  PROCEDURE msg_acked (the_id :  IN uid; from,
    to:  IN xtnded_uid);
  --gives an ack for a message
  PROCEDURE add_msg (the_len :  IN INTEGER;
    the_id :  IN uid; from, to :  IN xtnded_uid);
    the_txt :  IN msg_access; the_type :  IN msg_type);
  --gives a new message to the packet
  --manager to packetize.
  FUNCTION pkt_to_send (the_pkt :  OUT packet);
    RETURNS BOOLEAN;
  PROCEDURE packet_acked (the_uid :  IN uid;
    the_pos, the_tot :  IN INTEGER);
  --gives the id of a packet that has been ackd.
  PROCEDURE ack_this_pkt (the_uid :  IN uid;
    the_pos, the_tot :  IN INTEGER);
  --requests a packet acknowledgement
END PACKAGE packet_mgr;
```

The send_driver task has no entry points. It makes conditional entry calls on the call handlers for calls and responses to calls provided the packet_mgr has space. The packet_mgr has a small buffer to store message acknowledgements and these are sent on a priority basis. Thus the packet_mgr imposes no constraints on message acknowledgements. The send_driver, therefore, delivers all message acknowledgements to the packet_mgr.

The send driver also makes conditional entry calls to the net_tranceiver task. It delivers packets to this task and obtains packet acknowledgements, and packet acknowledgements it must send from this task.

```
TASK BODY send_driver IS
  ack_flag, msg_flag :  BOOLEAN := false;
  --if true implies call handler gave message or ack.
  the_hndlr :  ch_ptr;
  --holds current call handlr.
  the_bitbnd :  INTEGER; -- upper bound on bits
  the_pkbnd :  INTEGER; --upper bound on packets
  --call information
  the_uid :  uid;
  the_txt :  msg_access ; --pointer to message
  the_src, the_dst :  xtnded_uid;
  --ack information for messages
  ack_uid :  uid;
  ack_from; ack_to :  xtnded_uid;
  --packet information
  pkt_there :  BOOLEAN := FALSE ; --packet to send
  the_pkt :  PACKET ; --packet to send
  --packet acknowledgement information
  to_ack_flag, pkt_ack_flag :  BOOLEAN := FALSE
  packet_uid :  uid;
  packet_pos, packet_tot :  INTEGER;
BEGIN
  the_hndlr := typ_mgr_map.get_first;
  LOOP
  --1.  Find space for messages packet_mgr
  packet_mgr.how_many_free (the_pkbnd, the_bitbnd);
  IF NOT msg_flag THEN
    SELECT
      the_handlr.gvcr (the_bitbnd, the_uid, the_txt,
        the_src, the_dst, the_opt);
      IF the_bitbnd >0 THEN msg_flag := TRUE ENDIF;
    ELSE
      msg_flag := FALSE;
    END select;
  ENDIF;
  --2.  Find out if ack exists.
  IF NOT ack_flag THEN
    SELECT
      the_hndler.gvack (ack_uid, ack_from, ack_to);
      ack_flag := true;
    ELSE
      ack_flag := false;
    END SELECT;
  ENDIF;
  --3.  Act on messages or acks received.
  IF msg_flag THEN
    packet_mgr.add_msg (the_bitbnd, the_uid, the_txt,
      the_src, the_dst, the_opt);
  ENDIF;
```

```
   IF ack_flag THEN
     packet_mgr.msg_acked (ack_uid, ack_from, ack_to);
   ENDIF;
   --4.  Try to send a packet.
   IF NOT pkt.there THEN
     pkt_there := packet_mgr.pkt_to_send (the_pkt);
   THEN
   IF pkt_there THEN
     SELECT
       net_tranceiver.snd_data (the_pkt);
       pkt_there := false;
       ELSE
         null;
     END SELECT;
   ENDIF;
   --5.  Find out about packets acked or received.
   SELECT
     net_tranceiver.get_ack (packet_uid, packet_pos,
       packet_tot);
     pkt_ack.flag := TRUE;
   ELSE
     pkt_ack_flag := FALSE;
   END select;
   IF pkt_ack_flag THEN
     packet_mgr.packet_acked (packet_uid, packet_pos,
       packet_tot);
   ENDIF;
   SELECT
     net_tranceiver.give_ack (packet_uid, packet_pos,
       packet_tot);
     to_ack flag := TRUE;
   ELSE
     to-ack_flag := FALSE;
   END SELECT;
   IF to_ack_flag THEN
     packet_mgr.ack_this_pkt (packet_uid, packet_pos,
       packet_tot);
   ENDIF;
 END LOOP;
 END send_driver;
```

2.2.10.1.2 The Receive Task

The receive task has a packet manager that performs re- assembly of packets into messages. It also generates acks or naks for packets based on their correctness and the amount of packets buffer space available. The details of the packet manager are unspecified. The receive driver task polls the entries of the net tranceiver task that give out packets and accepts acks/nacks from the receive_driver. As mentioned earlier, the net tranceiver then forwards these responses to the send task which encapsulates then into outgoing packets. The advantage of leaving the packet manager unspecified is to let the link level protocol decisions be made within it.

```
PACKAGE rec_pkt_mgr IS
  FUNCTION give_packet (the_pkt :  IN packet) RETURNS
    pkt_info;
  --the manager receives a packet and returns an
  --ack or a nak.
  FUNCTION get_msg (the_ptr :  OUT msg_access;
    the_uid :  OUT uid; the_svc, the_dest :  OUT
      xtnded_uid;
    the_hndlr :  OUT chptr) RETURNS BOOLEAN;
  --returns a true if message is available
  FUNCTION get_ack (the_uid :  OUT uid; the-src, the_dest :
    OUT xtnded_uid; the_hndlr :  OUT ch_ptr)
    RETURNS BOOLEAN;
  --returns a null ack
  --else returns a uid.
END PACKAGE rec_pkt_mgr.
```

The receive driver (rec_driver) task is very similar to the send_driver task. The driver will not receive a packet from the net_tranceiver if its buffer pkt_answers is full. This buffer is an instantiation of the_buffer, a generic declaration.

```
TASK receive_driver IS
  --no entry points here.
END receive_driver;
TASK BODY receive_driver IS
  num_answers :  INTEGER CONSTANT :=
  pkt_answers IS NEW the_buffer (num_answers, pkt_info);
  the_pkt :  packet;
  the_aknk, to_go_aknk :  pkt_info;
  --packet and packet information buffers.
  rdy_aknk :  BOOLEAN := false;
  ack_flg :  BOOLEAN := FALSE;
  ack_uid :  uid;
  ack_src, ack_dst :  xtnded_uid;
  ack_hndlr :  ch_ptr;
  msg_uid :  uid;
  msg_src, msg_dst :  xtnded_uid;
  msg_hndler :  ch_ptr;
  msg_data :  msg_access;
  msg_flg :  BOOLEAN := false;
BEGIN
  LOOP
    IF NOT (pkt_answers.is_empty OR rdy_aknk) THEN
      to_go_aknk := pkt_answers.get_elem;
        rdy_aknk := TRUE
    ENDIF
    IF NOT pkt_answers.is_full THEN
      SELECT
        net_tranceiver.get_data (the_pkt);
        the_aknk := rec_pkt_mgr.give_packet
          (the_pkt);
```

This finishes the description of the RPC mechanism. It is not clear yet how call timeouts will be handled. At the present time, the notion of a roll_call computation is attractive. This will be described later in the design.

2.3 OBJECT STORAGE AND RETRIEVAL

There are two classes of object management: simple object management and stable object management (in the Lampson sense). Each class has different requirements and, therefore, must be treated on a different basis. The differences start at the kernel interface. This section first describes the interface for storage and retrieval. Next, the architectures of the simple storage is described, followed by the architecture of stable storage.

2.3.1 The Kernel Interface for Object Management

The kernel interface needs operations to read, write, and delete objects. Each of these operations is needed for simple and stable storage. Each of the operations returns a status to the user. The status for simple storage will differ from that for stable storage; this is the first visible difference to the user.

```
PACKAGE kernel IS

  --*put in all the type definitions and procedure
  --*specifications for the RPC interface as here.

  TYPE simpl_status IS
    (ready, not_available, does_not_exist, no_storage);
  --ready ==> operation completed successfully
  --not_available - ==> that the data has become
  --unavailable due to storage error.
  --does_not_exist ==> that there is no record of
  --the object on storage.
  --no_storage ==> no secondary storage

  TYPE stabl_status IS
    (ready, may_be_old, does_not_exist, no_storage);
  --ready ==> operation successful.
  --may_be_old ==> one copy readable.
  --does_not_exist ==> no record of object.
  --no_storage ==> no free space.

  TYPE obj_string IS ARRAY (RANGE <>) OF BOOLEAN;
  --a bit string that contains the object's data.
  --here follows the declarations of the procedures
  --at the kernel interface.
  --again the type uid must be supplied
  --in order to map to the type directory.
```

```
          pkt_answers.add_elem (the_aknk);
        ELSE
          NULL;
        END SELECT
      ENDIF
      IF rdy_aknk THEN
        SELECT
          net_tranceiver.put_acks (to_go_aknk);
          rdy_aknk := false;
        ELSE
          NULL;
        END SELECT;
      ENDIF;
      IF NOT msg_flg THEN
        msg_flg := rec_pkt_mgr.get_msg (msg_data,
          msg_uid, msg_src, msg_dest, msg_hndlr)
      ENDIF
      IF NOT ack_flg THEN
        ack_flg := rec_pkt_mgr.get_ack (ack_uid,
          ack_src, ack_dest, ack_hndlr);
      ENDIF
      --try to get the two call handlers
      IF msg_flg THEN
        SELECT
          msg_hndlr.gtcr (msg_uid, msg_data,
            msg_src, msg_dest);
          msg_flag := false;
        ELSE
          NULL;
        END SELECT;
      ENDIF;
      IF ack_flg THEN
        SELECT
          ack_hndlr.gtack (ack_uid, ack_svc,
            ack_dest);
        ELSE
          NULL;
        END SELECT;
      ENDIF;
    END LOOP;
  END receive_driver;
```

The packet_mgr and the pkt_rec_mgr handle all the internals of the line control protocols. These include buffer management, data checking, timing out of packets, retransmission of packets, sequencing, etc. This is a good approach because specifications of the line protocol can be done based on the designer's choice.

The number of rendezvous for an ack/nak to be sent to an incoming packet may seem excessive. It has been done this way to ensure that alternating layers in the kernel have either some entry point or no entry points. This technique prevents deadlock.

This finishes the description of the RPC mechanism. It is not clear yet how call timeouts will be handled. At the present time, the notion of a roll_call computation is attractive. This will be described later in the design.

2.3 OBJECT STORAGE AND RETRIEVAL

There are two classes of object management: simple object management and stable object management (in the Lampson sense). Each class has different requirements and, therefore, must be treated on a different basis. The differences start at the kernel interface. This section first describes the interface for storage and retrieval. Next, the architectures of the simple storage is described, followed by the architecture of stable storage.

2.3.1 The Kernel Interface for Object Management

The kernel interface needs operations to read, write, and delete objects. Each of these operations is needed for simple and stable storage. Each of the operations returns a status to the user. The status for simple storage will differ from that for stable storage; this is the first visible difference to the user.

```
PACKAGE kernel IS

  --*put in all the type definitions and procedure
  --*specifications for the RPC interface as here.

  TYPE simpl_status IS
    (ready, not_available, does_not_exist, no_storage);
  --ready ==> operation completed successfully
  --not_available - ==> that the data has become
  --unavailable due to storage error.
  --does_not_exist ==> that there is no record of
  --the object on storage.
  --no_storage ==> no secondary storage

  TYPE stabl_status IS
    (ready, may_be_old, does_not_exist, no_storage);
  --ready ==> operation successful.
  --may_be_old ==> one copy readable.
  --does_not_exist ==> no record of object.
  --no_storage ==> no free space.

  TYPE obj_string IS ARRAY (RANGE <>) OF BOOLEAN;
  --a bit string that contains the object's data.
  --here follows the declarations of the procedures
  --at the kernel interface.
  --again the type uid must be supplied
  --in order to map to the type directory.
```

```
  PROCEDURE get_obj (the_type:  IN uid;
    the_id:  IN xtnded_uid; the_obj:  OUT obj_string;
    the_status:  OUT simpl_status);

  PROCEDURE put_obj (the_type:  IN uid;
    the_id:  IN xtnded_uid;the_obj:  IN obj_string;
    the_status:  OUT simpl_status);

  PROCEDURE del_obj (the_typ:  IN uid;
    the_id:  IN xtnded_uid; the_status:  OUT
    simpl_status);

  --the above procedures define simple
  --object storage and retrieval.

  PROCEDURE stabl_get (the_typ:  IN uid;
    the_id:  IN xtnded_uid; the_obj:  OUT obj_string;
    the_status:  OUT stabl_status);

  PROCEDURE stabl_put (the_typ:  IN uid;
    the_id:  IN xtnded_uid; the_obj:  IN obj_string;
    the_status:  OUT stabl_status);

  PROCEDURE stabl_del (the_uid:  IN uid;
    the_id:  xtnded_uid; the_status:  OUT stabl_status);

  --the above procedures define stable object
  --storage and retrieval.

END PACKAGE kernel;
```

2.3.2 The Architecture of Storage

The above kernel calls are synchronous. Thus, a kernel procedure will not return until the operation has been completed. This presents the following problem. An entry call to a task in ADA does not achieve a rendezvous based on its parameters. Rather, the underlying scheduler will choose one of the calls waiting on the entry point in an arbitrary fashion. Thus, there needs to be a bond between the procedure instantiation of a request and the storage manager.

To achieve this, and to increase the parallelism in the kernel, the following architecture for the storage manager is suggested.

1. Each type will have two directories: one for simple storage and the other for stable storage. This separation will ensure that simple storage operations do not incur the penalty of performance that stable storage operations suffer.

2. Each directory will have associated with it sets of task instances. One each for a different object operation.

Thus, the cardinality of each set imposes a bound on the number of operations of each type for one object type at any given time. The pointer to an element of this set is returned to each call accepted. This element will be the task which asynchronously performs the operation and returns to the caller the result of the operation.

3. Each directory maps the extended uid of an object to one or more disc addresses. It is assumed here that the map is entirely in primary storage because reading in a part of a map from store is an overhead that can be modeled if desired. It must be remembered that stable directories must have two disc addresses per object or object version.

Figure 2-10 describes the sequence of actions in the kernel in response to a get_obj call at the kernel interface.

The next step is to describe the architecture of the simple and stable directories.

2.3.2.1 The Simple Directory

The simple directory is a manager of a set of objects of a given type. At the initiation of a type manager this directory is allocated a space on some disc so that a copy can be maintained on secondary storage.
The simple directory consists of the simple directory task and a mapping function. The latter maps extended uids of existing objects to disc addresses.

Associated with the directory are three sets of request_handlers: one each for the read, write and delete functions. The motivation behind a set of request handlers for each operation is to ensure that reads and deletes can go on even if writes are blocked due to a lack of disc space.

2.3.2.1.1 The Request Handlers

The request handler tasks are similar to the operation tasks used by type managers. They are allocated by the directory task and then exist as long as the directory tasks.

The request handlers first get a request from the directory, perform the request, and finally return the result to the kernel operation that made the request. After this they return to the directory for the next request.

2.3.2.1.2 The Mapping Manager

The mapping manager is an ADA package within the directory. Its primary function is to store the mapping of object extended uids to disc addresses. The mapping manager also maintains a table of the mapping from extended uid to disc addresses of new objects stored on secondary storage. Periodically the directory task will call the mapping manager and clear out the table of new entries.

The mapping manager at any time contains the entries for all object instances of a type. Thus, we are not concerning ourselves with the allocation of secondary storage to manage disc storage. However, a periodic writing of new entries to storage can ensure that the load of writing the directory entries is represented.

2.3.2.1.3 The Directory Task

The directory task is the main task in the simple directory. It receives requests, allocates them to request handler tasks, updates the map if needs be, and writes out the map to secondary storage.

2.3.2.2 The Stable Directory

The stable directory is very similar in architecture to the simple directory. The directory implements what Lampson calls a stable set. A stable storage of objects means that the directory that stores such objects must be stable. Thus, the storage of the map also must be to stable storage.

2.3.2.3 Consistency of Objects

The design, as outlined above, does not consider the consistency of objects. For example, if one is not careful, an object could be read and deleted simultaneously. The philosophy of ZEUS required that all access to objects occurs through its type manager. Further, the type uid is used to map to the directory. Thus, the access pointer is available to callers who supply the proper type uid.

In effect, the design leaves the responsibility of ensuring correct concurrency of access to the type manager. Errors in the type manager code may result in damage to the objects.

2.3.3 The Support Packages

There are two major support packages. The first of these is type_mgr_map which was described earlier. The second is the free_storage package which keeps track of free space on secondary storage.

2.3.3.1 The type_mgr_map Package

As in the case of call handlers, the type_mgr_map package must map type uids to simple directories and stable directories. This implies get, install, and put functions for each directory type.

Internally the package will have three tasks so that maximum parallelism can be obtained.

```
PACKAGE type_mgr_map IS

  --*Insert the calls that give access to call handler tasks.
```

```
    FUNCTION get_smd_ptr (type_uid:  IN uid) RETURNS
                          smd_ptr;

    PROCEDURE install_smd_ptr (typ_uid:  IN uid;
                               the_ptr:  IN smd_ptr);

    PROCEDURE delete_smd_ptr (typ_uid:  IN uid);

    --the above routines handle the simple directories
    --the pointer to a simple directory is
    --smd_ptr.

    FUNCTION get_sbd_ptr (typ_uid:  IN uid) RETURNS sbd_ptr;

    PROCEDURE install_sbd_ptr (typ_uid:  IN uid;
      the_ptr:  IN sbd_ptr);

    PROCEDURE delete_sbd_ptr (typ_uid:  IN uid);
    --the above routines handle the stable directories.

    --the pointer to a stable directory is
    --sbd_ptr.

  END PACKAGE type_mgr_map;
```

2.3.3.2 The free_storage Package

The free_storage package keeps track of secondary storage for simple objects. The calls to this package free or reserve storage blocks of arbitrary size. The users of this package request or release the starting address on secondary storage and the length of the block.

The package checks to ensure that the storage block being reserved or released is not in use, or in use respectively, at the time of the request.

```
  PACKAGE free_storage IS

    type_stor_status IS (free_success, reserve_success,
      free_fail, reserve_fail);

    FUNCTION free_block (the_addr:  IN secst_addr;
      the_length:  IN INTEGER) RETURNS stor_status;

    FUNCTION get_block (the_addr:  OUT secst_addr;
      the_length IN INTEGER) RETURNS stor_status;

    PROCEDURE bad_block (the_addr:  IN secst_addr;
      the_length:  IN INTEGER);

    --the secst.addr type is an address on
    --secondary storage.

  END PACKAGE free_storage;
```

2.3.4 The stable_free Package

The stable_free package keeps track of secondary storage for stable objects. Calls to this package free or reserve storage blocks of arbitrary size. The users of this package free or reserve twin blocks of secondary storage of arbitrary size on different units.

If one of the blocks at which an object is stored decays then both blocks are returned and new ones, if available, are obtained from this package.

```
PACKAGE stable_free IS

  TYPE stbl_avail IS (freed, reserved, freed_fail,
    reserved_failed);

  FUNCTION free_block (the_addr1, the_addr2:  IN secst_addr;
    the_length:  IN INTEGER) RETURNS stbl_avail;
  FUNCTION get_block (the_addr1, the_addr2:  OUT secst_addr;
    the_length:  IN integer) RETURNS stbl_avail;

  PROCEDURE bad_block (the_addr1, the_addr2:  IN secst_addr;
    the_length:  IN integer);

END PACKAGE stable_free;
```

A generic package can be used to specify both types of free storage.

2.3.5 The Request Handler Tasks

The request handler tasks are specified as task types. Each type performs different operations for the directory task. The organization of this section is as follows. In section 2.3.4.1, the specifications of the task types for the simple directory are given. This is followed in section 2.3.4.1.1 by the task body specification of the put simple object function as an example of the interactions between the task and the rest of the system. In section 2.3.4.2, the specifications of the task types for the stable directory are given. Again the specification of the put stable object is specified as an example in section 2.3.4.2.1.

The request handlers make a number of entry calls to the directory. Each handler has two entry points. The first is used by the directory to initialize the request handler. The second is used by the kernel procedure to obtain the result of the operation it requested.

The request handler types are given names whose first phrase specifies the type of directory it works with, and whose second phrase describes the type of function it will perform. Access pointers to these types have type names suffixed by 'ptr'.

2.3.5.1 The Simple Directory Request Handlers

The simple directory request handlers belong to one or more of these task types: smp_get, smp_put and smp_del. Each of these types have the entry points get_init and get_result for initialization and for function result.

```
TYPE smp_get;

TYPE smp_get_ptr IS ACCESS smp_get;

TASK TYPE smp_get IS

  ENTRY get_init (myself:  IN smp_get_ptr;
    the_dir:  IN smd_ptr);

  --the task contains a self_referential pointer
  --so that it can pass it to the directory
  --when it requests a function.  The smd_ptr

  --gives the task instance the pointer to
  --the directory it is connected to.

  ENTRY get_result (the_object:  OUT obj_string;
    the_status:  OUT simpl_status);

  --this gives the requesting kernel procedure
  --the object it requires.

END TASK smp_get;
```

The smp_get task performs the get simple object function. While a self referential pointer may not be exactly kosher, it is useful in that the directory task does not need a mapping from some other identifier to the pointer of the task.

```
TYPE smp_put;

TYPE smp_put_ptr IS ACCESS smp_put;

TASK TYPE smp_put IS

  ENTRY get_init (myself:  IN smp_put_ptr;
    the_dir:  IN smd_ptr);

  ENTRY get_result (the_status:  OUT simpl_status);

  --the object will be passed when the
  --task makes its entry call to the
  --directory.

END TASK smp_put;
```

The smp_put task performs the put simple object function. Its specification is very similar to the smp_get task.

The smp_del task performs the delete simple object function. Its specification follows.

```
TYPE smp_del;

TYPE smp_del_ptr IS ACROSS smp_del;

TASK TYPE smp_del IS

  ENTRY get_init (myself:  IN smp_del_ptr;
    the_dir:  IN smd_ptr);

  ENTRY get_result (the_status:  OUT simpl_status);

END TASK smp_del;
```

2.3.5.1.1 The Function smp_put

The smp_put task performs the following sequence of actions after initialization. A graphic for the actions can be seen in Figure 2-11.

1. Obtains the object to be written along with its secondary storage address, if this is a replacement.

2. If the object is not being replaced, the task calls the free_storage package to get a secondary storage address for the object.

3. It uses the secondary storage address to obtain the pointer to the disc controller task.

4. It initiates a write operation and, on completion of this operation, it calls the directory to pass back the new secondary storage address, if necessary.

5. Returns the status of the operation back to the kernel procedure.

```
TASK BODY smp_put IS

  self_ptr:  smp_put_ptr;
  dir_ptr:  smd_ptr;

  --pointers to self and directory.

  the_obj_id:  xtnded_uid;
  the_obj:  obj_string;
```

```
  --object string and identifier.

  the_addr:  secst_addr;

  --secondary storage address.

  buf_status:  free_storage.stor_status;

  --secondary storage status.

  op_status:  simpl_status;

  --operation status;

  log_device:  d_ctl_ptr;

  --pointer to the device controller

BEGIN

  ACCEPT get_init (myself_ IN smp_put_ptr;
      the_dir:  IN smd_ptr) DO
    self_ptr:= myself;
    dir_ptr:= the_dir;
  END

LOOP
  --get operation to be performed.
  dir_ptr.heres_put (self_ptr, the_obj_id, the_obj,
    the_addr, the_length);
  op_status := ready;
  IF the_addr.secst_dev_ptr = nil THEN
    --get a block of storage.
    the_length := obj_length (the_obj),
    buf_status := free_storage.get_block (the_addr,
      the_length);
    IF buf_status = reserve_fail THEN
      --no operation possible, full secondary storage.
      op_status := no_storage;
    ENDIF;
  ENDIF;

    IF op_status = ready THEN
      log_device := device_map.get_dev (the_addr.secst_dev);
        log_device.write (the_addr.dev_addr,
        the_obj, the_length, d_state);
      --perform the i/o
      IF d_state /= complete THEN
        op_status := not_available;
        free_storage.bad_block (the_addr, the_length);
        --tell free_storage about bad sectors.
      ENDIF;
    ENDIF;
```

```
        IF op_status = ready THEN
          dir_ptr.put_dir (the_obj_id, the_addr, the_length);
          --set up the directory entry for the object.
        ENDIF
        --give result back to the caller.
        ACCEPT get_result (the_status:  OUT simpl_status) DO
          the_status := op_status;
        END;
      END LOOP;
    END TASK smp_put;
```

This task will return a not_available status to its caller if the actual write to secondary storage did not complete. This is in keeping with the philosophy of the simple put operation.

2.3.5.2 The Stable Directory Request Handlers

The stable directory request handlers belong to one or more of these task types: stb_get, stb_put, and stb_del. Again each of these types have the entry points get_init and get_result for initialization and for function result. The tasks are specified below.

```
TYPE stb_get;

TYPE stb_get_ptr IS ACCESS stb_get;

TASK TYPE stb_get IS

  ENTRY get_init (myself:  IN stb_get_ptr:
    the_dir:  IN sbd_ptr);

  ENTRY get_result (the_object:  OUT obj_string;
    the_status:  OUT stabl_status);
END TASK stb_get;
```

This task type performs the stable get function as defined by Lampson. Again the result returns the object and the status of the object.

```
TYPE stb_put;

TYPE stb_put_ptr IS ACCESS stb_put;

TASK TYPE stb_put IS

  ENTRY get_init (myself:  IN stb_put_ptr;
    the_dir:  IN sbd_ptr);

  ENTRY get_result (the_status:  OUT stabl_status);

END TASK stb_put;
```

This routine handles the stable put operation for objects.

```
TYPE stb_del;

TYPE stb_del_ptr IS ACCESS stb_del;

TASK TYPE stb_del IS

  ENTRY get_init (myself:  IN sbt_del_ptr;
    the_dir:  IN sbd_ptr);

  ENTRY get_result (the_status:  OUT stabl_status);

END TASK stb_del;
```

This routine handles stable delete operations.

2.3.5.2.1 The Stable Put Operation

The stable_put operation is derived from Lampson's stable_put operation. In effect, it is a combination of the insert and replace operations in the stable set of Lampson.

The operation has the following sequence.

1. The object, its id, and the two secondary storage address (if the object already exists) are obtained.

2. The careful puts to each address must be done one after the other.

3. The directory is passed back the secondary storage address.

4. The kernel procedure is returned the status of the operation.

The careful_put is specified as a function that attempts to write to an address a fixed number of times. If it fails, it returns a failure mode back to the stable put. This failure of the careful put results in invocations of free storage to return the block and to obtain a new block. If no block is available, the failure mode is returned to the kernel procedure.

```
TASK BODY stb_put IS

  self_ptr:  stb_put_ptr;
  dir_ptr:  sbd_ptr;
  --pointers to self and directory.

  the_obj_id:  xtnded_uid;
  the_obj:  obj_string;
  --object string and identifier;

  the_addr1, the_addr2:  secst_addr;
  --secondary storage
```

```
  buf_status:  stable_free.stbl_avail;
  op_status:  stabl_status;
  --secondary storage and operation status

  care_put_count:  INTEGER CONSTANT := 10;
  cp1, cp2:  BOOLEAN;
  --careful put tries to write at most
  --care_put_count times
  --cp1, cp2 are flags to indicate success
  --of careful_put operations.

BEGIN

  ACCEPT get_init (myself:  IN stb_put_ptr;
    the_dir:  IN sbd_ptr) DO
    self_ptr := myself;
    dir_ptr := the_dir;
  END;
  LOOP
    --get operation to be done
    dir_ptr.heres_sput (self_ptr, the_obj_id, the_obj,
      the_addr1, the_addr2, the_length);
    op_status := ready;
    IF the_addr1.secst_dev_ptr = nil THEN
      --get a block of storage.
      the_length := obj_length (the_obj);
      buf_status := stable_free.get_block (the_addr1,
        the_addr2, the_length);
      IF buf_status = reserved_failed THEN
        --no operation possible
        op_status := no_storage;
      ENDIF;
    ENDIF;

    --try to perform the stable put operation

    cp1 := FALSE; cp2 := false;

    WHILE (NOT (cp1 AND cp2) and op_status := ready) LOOP
      cp1 := false; cp2 := false;
      cp1 := careful_put (the_obj, the_length,
             the_addr1, care_put_count);
      cp2 := careful_put (the_obj, the_length,
             the_addr2, care_put_count);

      IF NOT (cp1 and cp2) THEN

        --return the bad blocks
        stable.free_bad_block (the_addr1, the_addr2,
          the_length);
        --get new blocks
        buf_status := stable_free.get_block (the_addr1,
```

```
              the_addr2, the_length);
            IF buf_status = reserve_failed THEN
              op_status := no_storage;
            ENDIF;
          ENDIF;
        END LOOP;
        IF op_status = ready THEN
          dir_ptr_put_dir (the_obj_id, the_addr1,
            the_addr2, the_length);
        ENDIF;
        --return status to caller
        ACCEPT get_result (the_status:  OUT stabl_status) DO
          the_status := op_status;
        END;
      END LOOP;
    END TASK stb_put;
```

One possible flaw in the code is that bad blocks are reported before new blocks are obtained. However, a failure here will occur before a directory change and, thus, the good copy may be recovered. The smp_put routine could be modified to do a repeated put, if so desired. Finally, an upper limit of attempts to write to secondary storage could be used to report bad secondary storage.

2.3.6 The Directory Package

The directory packages store information about the mapping from xtnded_uid to secondary storage address. Thus, they must have operations to enter, access, and delete mappings. In addition, the package should have a function to obtain blocks of map entries from secondary storage. In this design, however, we assume that the full directory mapping is constructed at host start-up time and is updated in memory during the system operation. This approach may not be realistic but it makes the design simpler.

There is one generic package specification and two instances of it. One of them is contained in the simple directory task body and is called smp_map, while the other is contained in the stable directory task body and is called the stb_map. But before this generic package is specified, the types secst_addr, smp_dir_entry, and stb_dir_entry are defined.

2.3.6.1 Some Basic Types

The type secst_addr has been used in preceding sections. It specifies a secondary storage address and consists of two parts: a device address and the address on that device.

```
    TYPE secst_addr IS
      RECORD
        secst_dev:  device_name;
        dev_addr:  device_location;
      END record;
```

The types device_name and device location are assumed to have been specified.

The types smp_dir_entry and stb_dir_entry differ only in the number of secondary storage addresses that are provided.

```
TYPE smp_dir_entry IS
  RECORD
    the_addr:  secst_addr;
    the_length:  INTEGER;
  END;

TYPE stb_dir_entry IS
  RECORD
    the_addr1, the_addr2:  secst_addr;
    the_length:  INTEGER
  END
```

2.3.6.2 The package specification

```
generic

  TYPE key IS LIMITED private;

  TYPE direntry IS LIMITED private;

package the_map IS

  FUNCTION get_entry (the_key:  IN key) RETURNS
    direntry;
  --returns null for absent keys.
  PROCEDURE put_entry (the_key:  IN key; the_entry:  IN
    direntry);
  --replaces new entry.
  FUNCTION del_entry (the_key:  IN key) returns
    direntry;
  --access, enter and deletes entries

END PACKAGE the_map;
```

2.3.7 The Directory Task

The directory tasks for both simple and stable directories have the same functions. These functions are:

1. To initialize the get, put, and delete request handlers.

2. To service kernel procedure requests and to form these requests to the appropriate request handlers.

3. To handle requests to the directory map data structures.

To accomplish this, the directory task has a number of entry points. The two major sets of entry points are for the reception of kernel calls and the reception of directory map calls. To link up the kernel procedure that makes a request and a request handler, the directory task has a third set of entries for available directory tasks. The entry points of the directory task and their interactions are graphically illustrated in Figure 2-12.

In effect, the directory acts as a monitor that schedules requests to secondary storage and protects the directory.

2.3.7.1 The Interface of the Simple Directory Task

```
TYPE smp_dir;

TYPE smd_ptr IS ACCESS smp_dir;
TASK TYPE smp_dir IS

  ENTRY get_obj (the_obj_id:  IN xtnded_uid; get_hndlr:  OUT
    smp_get_ptv);
  ENTRY put_obj (the_obj_id:  IN xtnded_uid; out_obj:  IN
    obj_sring; put_hndlr:  OUT smp_put_ptr);
  ENTRY del_obj (the_obj_id:  IN xtnded_uid; del_hndlr:
    OUT smp_del_ptr);

  --the above entries handle kernel procedures.
  ENTRY get_dir (the_obj_id:  IN xtnded_uid; the_addr:  OUT
    secst_addr; the_length:  OUT INTEGER);
  ENTRY put_dir (the_obj_id:  IN xtnded_uid; the_addr:  IN
    secst.addr; the_length:  IN INTEGER);
  ENTRY del_dir (the_obj_id:  IN xtnded_uid);

  --the above entries allow calls to the directory
  --data structure.

  ENTRY heres.get (the_id:  IN smp_get_ptr; obj_uid:  OUT
    xtnded_uid; the_addr:  OUT serst_addr; the_length:  OUT
    INTEGER);

  ENTRY heres_put (the_id:  IN smp_put_ptr; obj_uid:  OUT
    xtnded_uid; the_obj:  OUT obj_string;
    the_addr:  OUT secst_addr; the_length:  OUT INTEGER);

  ENTRY heres_del (the_id:  IN smp_del_ptr; obj_uid:  OUT
    xtnded_uid; the_addr:  OUT secst_addr;
    the_length:  OUT INTEGER);

  --the above are entry points by which the
  --request handlers make themselves
  --available.  the_obj_id parameters is superflous.

  ENTRY get_init (the_id:  IN smd_ptr;
    get_count, put_count, del_count:  IN INTEGER);
```

```
--thru this entry point the directory is initialized
--it obtains its own identity, and the count
--of the number of request handlers it can start.

END TASK smp_dir
```

2.3.7.2 The Simple Directory Task Body

The simple directory task schedules requests from the kernel procedures. However, the simple directory task cannot accept a procedure request until a request handler of the corresponding request is available. Thus, for example, a get_obj entry call is accepted only when the number of entry calls at the heres_get entry point is greater than zero.

```
TASK BODY smp_dir IS

  the_id:  xtnded_uid;
  the_obj:  obj_string;
  the_addr:  secst_addr;
  the_length:  INTEGER;
  --object being transferred and its attributes

  num_gets, num_puts, num_dels:  INTEGER;
  --number of request handlers

  this_dir:  smd_ptr; --self reference
  the_get:  smp_get_ptr;
  the_put:  smp_put_ptr;
  the_del:  smp_del_ptr;
  --pointers to the request handlers (temporary)

  smp_map IS NEW the_map (xtnded_uid, smp_dir_entry);
  --instantiate the mapping package

  blk_data:  smp_dir_entry;
  --directory map

BEGIN
  --get initialized
  ACCEPT get_init (the_id:  IN smd_ptr; get_count,
      put_count, del_count:  IN INTEGER) DO
    this_dir := the_id;
    num_gets := get_count;
    num_puts := put_count;
    num_dels := del_count;
  END;

  --start up request handlers
  FOR i IN 1..numgets LOOP
    the_get := new smp_get;
    the_get.get_init (the_get, this dir);
  END
```

```
  FOR i IN 1..num_puts LOOP
    the_put := new smp_put;
    the_put.get_init (the_put, this_dir);
  END

  FOR I IN 1..num_dels LOOP
    the_del := new smp_del;
    the_del.get_init (the_del, this_dir);
  END

LOOP --loop forever

  SELECT
    --handle kernel entries

    WHEN E'heres_get > 0
      ACCEPT get_obj (the_obj_id:  IN xtnded_uid;
        get_hndlr:  OUT smp_get_ptr) DO
        blk_data := smp_map.get_entry (the_obj_id);
        --the directory entry may contain a
        --null address and zero length
        ACCEPT heres_get (the_id:  IN smp_get_ptr;
        obj_uid:  IN xtended_uid; the_addr:  OUT
        secst_addr; the_length:  OUT INTEGER) DO
          get_hndlr := the_id;
          obj_uid := the_id;
          the_addr := blk_data.the_addr;
          the_length := blk_data.the_length;
        END;
      END;

    OR
      WHEN E' heres_put > 0
        ACCEPT put_obj (the_obj_id:  IN xtnded_uid;
          out_obj:  IN obj_string; put_hndlr:  OUT
          smp_put_ptr) DO
          blk_data := smp_map.get_entry (the_obj_id);
          ACCEPT heres_put (the_id:  IN smp_put_ptr;
          obj_uid:  OUT xtnded_uid; the_obj:  OUT
          obj_string; the_addr:  OUT secst_addr;
          the_length:  OUT INTEGER) DO
            put_hndlr := the_id;
            the_obj := out_obj;
            obj_uid := the_obj_id;
            the_addr := blk_data.the_addr;
            the_length := blk_data.the_length;
          END;
        END;

    OR
      WHEN E' heres.del > 0
        ACCEPT del_obj (the_obj_id:  IN xtnded_uid;
          del_hndlr:  OUT smp_del_ptr) DO
```

```
          blk_data := smp_map.get_entry (the_obj_id)
          ACCEPT heres_del (the_id:  IN smp_del_ptr;
            obj_uid:  OUT xtnded_uid; the_addr:  OUT
            secst_addr; the_length:  OUT INTEGER) DO
            del_hndlr := the_id;
            obj_uid := the_obj_id;
            the_addr := blk_data.the_addr;
            the_length := blk_data.the_length;
          END;
        END;

      OR
        ACCEPT get_dir (the_obj_id:  IN xtnded_uid;
          the_addr:  OUT secst_addr, the_length:  OUT
          INTEGER) DO
          blk_data := smp_map.get_entry (the_obj_id);
          the_addr := blk_data.the_addr;
          the_length := blk_data.the_length;
        END;

      OR
        ACCEPT put_dir (the_obj_id:  IN xtnded_uid;
          the_addr:  IN secst_addr; the_length:  IN
          INTEGER) DO
          blk_data.the_addr := the_addr;
          blk_data.the_length := the_length;
          smp_map.put_entry (the_obj_id; blk_data);
        END;


      OR
        ACCEPT del_dir (the_obj_id :  IN xtnded_uid) DO
          blk_data := smp_map.dle_entry (the_obj_id);
        END;
      END SELECT;
    END LOOP;
  END TASK smp_dir;
```

2.3.7.3 The Stable Directory Task Interface

The stable directory task is specified here. The body of the task is identical in structure and spirit to the simple directory task and, therefore, is not defined.

```
TYPE sbd_dir;

TYPE sbd_ptr IS ACCESS sbd_dir;

TASK TYPE sbd_dir IS

  ENTRY get_obj (the_obj_id:  IN xtnded_uid;
    get_hndlr:  OUT stb_get_ptr);
```

```
ENTRY put_obj (the_obj_id:  IN xtnded_uid;
  out_obj:  IN obj_string; out_hndlr:  OUT
  stb_put_ptr);
ENTRY del_obj (the_obj_id:  IN xtnded_uid;
  del_hndlr:  OUT stb_del_ptr);

--the above entries handle kernel procedures.

ENTRY get_dir (the_obj_id:  IN xtnded_uid;
  the_addr1, the_addr2:  OUT secst_addr; the_length:
  OUT integer);

ENTRY put_dir (the_obj_id:  IN xtnded_uid;
  the_addr1, the addr2:  IN secst_addr; the_length:
  IN INTEGER);
ENTRY del_dir (the_obj_id:  IN xtnded_uid);

--these entries allow calls to the directory.

ENTRY heres_get (the_id:  IN stb_get_ptr; obj_uid:  OUT
  xtnded_uid; the_obj:  OUT obj_string; the_addr1,
  the_addr:  OUT secst_addr; the_length:  OUT INTEGER);
ENTRY heres_del (the_id:  IN stb_del_ptr; obj_uid:  OUT
  xtnded_uid; the_addr1, the_addr2:  OUT secst_addr;
  the_length:  OUT INTEGER);

--request handler service points.

ENTRY get_init (the_id:  IN sbd_ptr;
  get_count, put_count, del_count:  IN INTEGER);

END TASK sbd_dir;
```

2.3.8 The Storage Controllers

The storage controllers consist of a set of tasks each of which manages a secondary storage device. To map from device addresses which are device specific to the controllers, there is a controller_map package. This controller_map package is an instance of the generic package, the_map, with the types device_name as the key and d_ctl_ptr (device controller pointer) as the entry.

The specification of the device controller task is as follows:

```
TYPE d_state IS (complete, failed, bad_surface);

TYPE d_ctl;

TYPE d_ctl_ptr IS ACCESS d_ctl;

TYPE d_ctl IS
```

```
    ENTRY read (the_addr:  IN device_location;
      the_length:  IN INTEGER; the_obj:  OUT obj_string,
      the_stale:  OUT d_state);

    ENTRY write (the_addr:  IN device_location;
      the_length:  IN INTEGER; the_obj:  IN obj_string;
      the_state:  OUT d_state);

    ENTRY done;

    FOR done USE AT 8#---#;
    --interrupt

  END TASK d_ctl;
```

2.3.9 Kernel Procedures

The kernel procedures are very simple. They first obtain a pointer to the proper type directory. Then they request the directory to perform the function. The directory returns to them the request handler assigned to their request. Finally, the procedure asks the request handler for the result. On getting the result the procedure terminates.

2.4 SEQUENCE NUMBER GENERATION

Sequence number generation has to do with the generation of unique identifiers and extended unique identifiers for the ZEUS system. The functions provided by this part of the kernel include:

1. New uid generation.

2. Formation of an extended uid.

3. Giving out components of an extended uid.

In an earlier section of this document, the type definitions uid and xtnded_uid were specified. Here, we first describe each of these types and the motivation for the fields inside them. Then, the architecture required to service the kernel procedures is specified.

2.4.1 Identifying Objects in ZEUS

The unique identifier is a name for an object in the ZEUS system. The object may be temporary such as a call, it may be semi-permanent such as an instance of a type, or it may be permanent such as a type manager or type/type manager.

Only temporary objects can be identified solely by a unique identifier, all other objects must have an extended unique identifier.

2.4.1.1 The Unique Identifier (uid)

The unique identifier in ZEUS consists of three fields. The first is a host field which identifies the host at which the uid was generated. The second field is an incarnation field, while the third is the sequence field. The incarnation field is generated in a distributed fashion by the set of hosts in the ZEUS system. The algorithms in this section describe the process of generation. The sequence field allows a range of uids to be generated for each value of the incarnation field.

To recapitulate the uid is defined as

```
TYPE incrnge IS RANGE 0..(2**32-1);
TYPE seqrnge IS RANGE 0..(2**22-1);
TYPE host_id IS RANGE 0..1023;

--definitions for the host identifiers, incarnation
--and sequence fields of the uid.

TYPE uid IS
  RECORD
    origin_host:  host_id;
    incarnation:  incrnge;

    sequence:  seqrnge;
  END RECORD;
```

2.4.1.2 The Extended Unique Identifier (xtnded_uid)

The extended uid allows an object to be placed in its proper perspective. An object in ZEUS has a type associated with it. Thus, it is an instance of a type. A type name must be unique and, therefore, each type name is a uid. The instance of an object type has to be unique within that type. It is simpler, and more efficient, to use the kernel to generate a unique instance name as a uid, rather than to have the type managers generate unique identifiers. Within an instance of an object there may be more than one version. Thus, the extended uid has a version uid field. Again, the same reasoning justifies the usage of a uid instead of a special unique number.

Thus the xtnded_uid type is defined as

```
TYPE xtnded_uid IS
  RECORD
    host_hint:  host_id;
    typeuid, instanceuid, versionuid:  uid;
  END RECORD;
```

The previous discussion did not describe the host hint field. This field is a guess at where an object is currently located. This field is changed when the object moves from one host to another. When an object that references a moved object discovers its new location it updates the host hint in the xtnded_uid.

2.4.1.3 The Visible Types

The uid and xtnded_uid types are already visible. The host_id type must be visible outside the kernel, too. This is to allow the host_hint field in the xtnded_uid to be modified.

2.4.2 The Kernel Procedures

There are two types of kernel procedures. The first builds uids and xtnded uids, the second provides access to the components of an extended uid. The procedures are defined as part of the kernel specification which is continued here.

```
PACKAGE kernel IS

  --put in the RPC and storage kernel calls
  --and type definitions.

  TYPE host_id IS limited private;

  FUNCTION get_uid RETURNS uid;

  FUNCTION build_xt (the_host:  IN host_id;
    the_type, the_instance, the_version) RETURNS
    xtnded_uid;

  FUNCTION give_host_hint (the_in:  IN xtnded_uid)
    RETURNS host_id;

  FUNCTION change_hint (the_id:  IN xtnded_uid;
    new_hint:  IN host_id) RETURNS xtnded_uid;

  --get_uid is the unique number generation
  --function.

  --build_xt constructs an extended uid given
  --all of its components.

  --give_host_hint returns the host hint of the
  --extended uid.

  --change_hint modifies the host_hint
  --as directed.

END PACKAGE kernel;
```

2.4.3 The Architecture

The architecture to generate sequence numbers is very simple. Its kernel procedures, other than get_uid, do not need to wait. This is because they are all operations on limited private types that are defined inside the kernel.

Other than the kernel procedures, there are two tasks. The first is a monitor that allocates the incarnation and sequence fields for the current incarnation number value. This task calls the second task when it needs a new incarnation number.

The second task accesses the net tranceiver task to obtain new incarnation numbers, and to handle the starting up of failed hosts in the ZEUS system. These hosts need to be allocated the current incarnation number and a priority number. The priority numbers order the active hosts in the system. This is done to ensure that generation of incarnation numbers is resilient. The interaction between the uid_monitor and the uid_generation tasks is shown in Figure 2-13.

The kernel function, get_uid, is specified first, followed by the two tasks, uid_monitor and uid_generation. The other kernel procedures are not specified. To place the uid generation task in its proper perspective, the portion of the net_tranceiver task that handles the uid generation is described here.

2.4.3.1 The get_uid Function

The get_uid function makes available the next uid for the requestor.

```
FUNCTION get_uid RETURN uid IS
  this_uid: uid;
begin
  uid_monitor.get_uid (this_uid);

  RETURN this_uid;
END get_uid;
```

2.4.3.2 The uid_monitor Task

The uid_monitor task has two loops. The outer loop does not terminate, and first obtains a new incarnation number. It then enters the inner loop. This loop issues uids for the range of the current incarnation number and then exits back to the outer loop to get the next incarnation number. It is possible that the uid_generation task may have passed a new incarnation number to the uid_monitor before the previous range terminated. The uid_monitor has two entry points get_uid and get_inc.

```
TASK uid_monitor IS

  ENTRY get_uid (the_uid:  OUT uid);
  ENTRY get_inc (the_ind:  IN incrnge);

END TASK uid_monitor;

TASK BODY uid_monitor IS
```

```
  seq_bnd:  seqrnge CONSTANT := 2**22 -1;
  cur_seq:  seqrnge;
  --sequence number bounds.

  cur_inc:  incrnge;
  --current incarnation number;

  this_host:  host_id CONSTANT := ______;

BEGIN

  LOOP
    --request for a new incarnation number
    uid_generation.start_uid_req (cur_inc);
    ACCEPT get_inc (the_inc:  IN incrnge) DO
      cur_ind := the_inc;
    END;
    cur_seq := 0;

    WHILE (cur_seq < seq_bnd) OR (E'get_inc > 0) LOOP
      --new incarnations get priority over uids
      SELECT
        --new incarnation from uid_generation task
        ACCEPT get_inc (the_inc:  IN incrnge) DO
          cur_inc := the_inc;
        END;
        cur_seq := 0;

      OR
        --uid request from kernel procedure.
        WHEN E'get_inc = 0 ==>
        ACCEPT get_uid (the_uid:  OUT uid) DO
          the_uid.origin_host := this_host;
          the_uid.incarnation := cur_inc;
          the_uid.sequence := cur_seq;
```

PACKAGE KERNEL IS

RPC FUNCTIONS	STORAGE FUNCTIONS	UID GENERATION
MAKE_CALL	GET_OBJ	GET_UID
MAKE_RESP	GET_STABL_OBJ	BUILD_XT
KILL_CALL	PUT_OBJ	GIVE_HOST_HINT
KEEP_CALL	PUT_STABL_OBJ	CHANGE_HINT
GET_CALL	DEL_OBJ	
GET_RESP	DEL_STABL_OBJ	
C_STATUS		
R_STATUS		
VISIBLE TYPES	**VISIBLE TYPES**	**VISIBLE TYPES**
UID (P)	XTENDED_UID	UID
XTENDED_UID	OBJ_STRING	XTENDED_UID
MESSAGE	OBJ_STATUS	HOST_HINT
CALL_STATUS		
RESP_STATUS		
DEL_OPTION		

END PACKAGE KERNEL;

Figure 2-1 Kernel Interface

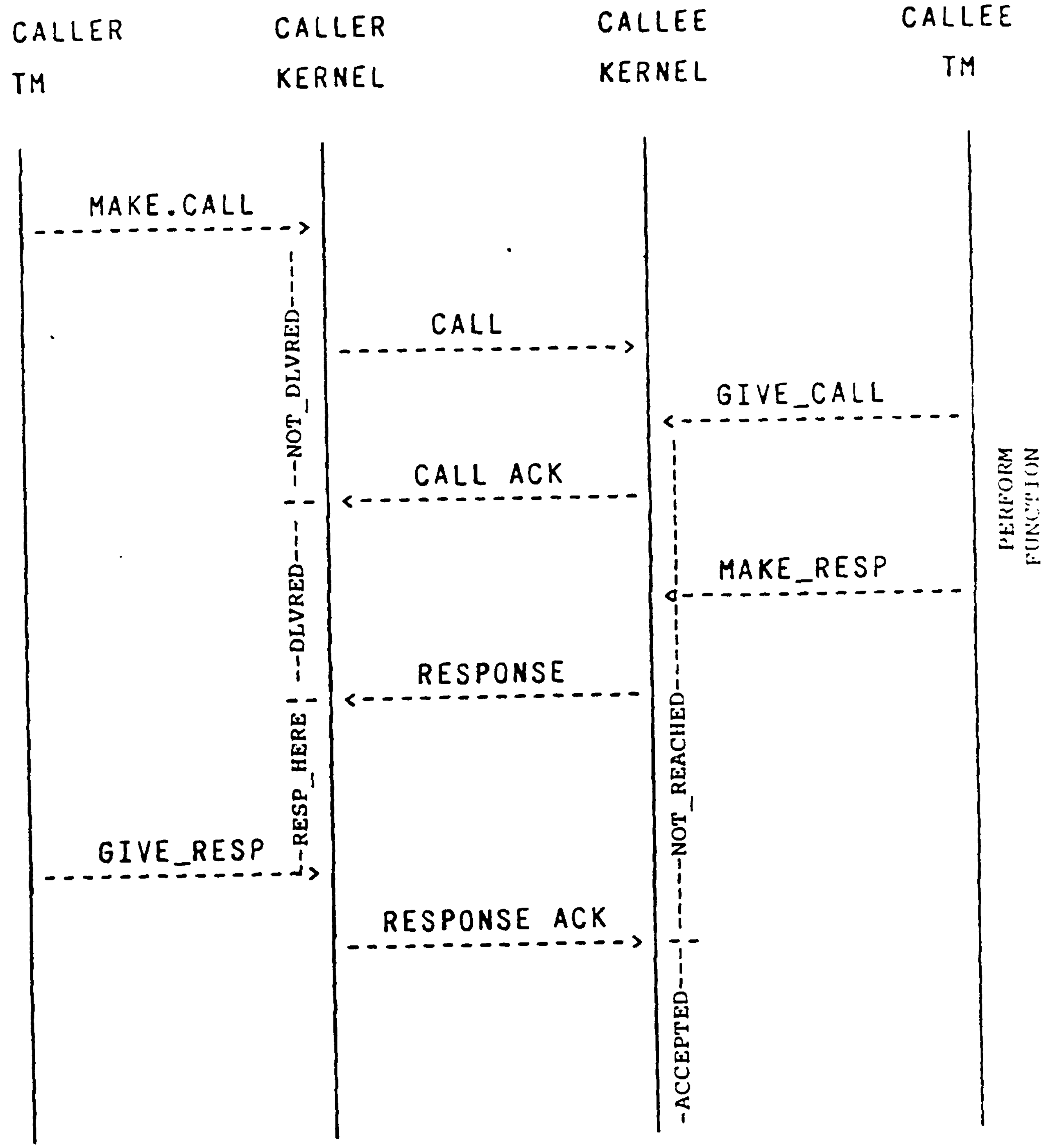


Figure 2-2 Messages Exchanged for the RPC Protocol

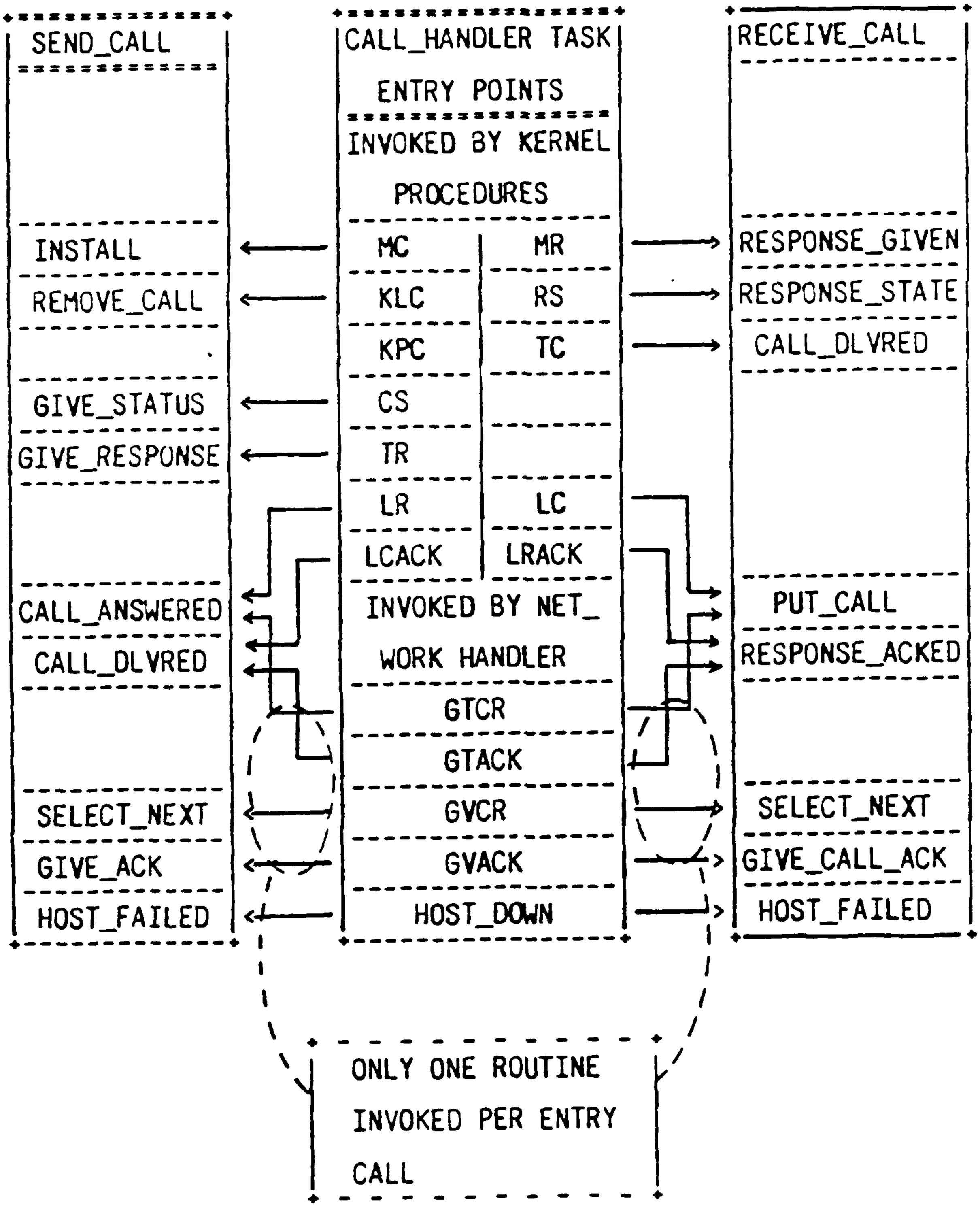


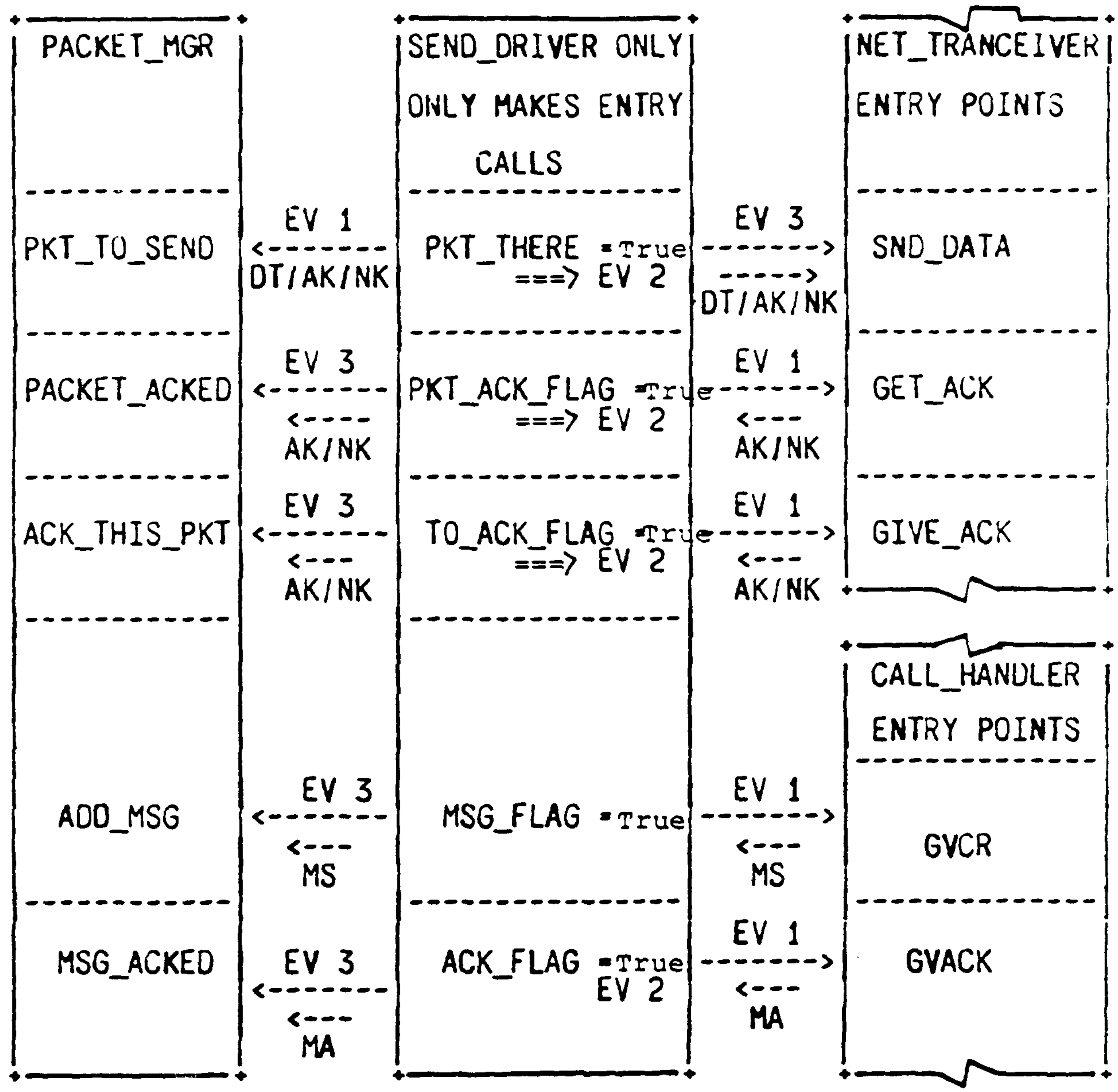
Figure 2-3 Call Handler Component Interaction

ENTRY NAME	CALLED BY	PURPOSE	IMPACT ON COMPONENTS	
			SEND_CALL	RECEIVE_CALL
MC	MAKE_CALL	SEND CALL	INSTALL	
MR	MAKE_RESP	SEND RESPONSE		RESPONSE_GIVEN
KLC	KILL_CALL	DELETE CALL	REMOVE_CALL	
KPC	KEEP_CALL	RETAIN CALL		
RS	R_STATUS	RESPONSE STATUS		RESPONSE_STATUS
CS	C_STATUS	CALL STATUS	GIVE_STATUS	
TC	GET_CALL	OBTAIN CALL		CALL_DLVRED
TR	GET_RESP	GET RESPONSE	GIVE_RESP	
LC	MAKE_CALL	LOCAL CALL		PUT_CALL
LR	MAKE_RESP	LOCAL RESPONSE	CALL_ ANSWERED	
LCACK	GET_CALL	LOCAL CALL ACK	CALL_ DLVRED	
LRACK	GET_RESP	LOCAL RESP ACK		RESPONSE_ACKED

Figure 2-4 Entry Points to the Call Handler Invoked by Kernel Procedures

ENTRY NAME	CALLED BY	PURPOSE	IMPACT ON COMPONENTS	
			SEND_CALL	RECEIVE_CALL
GTCR	RECEIVE_DRIVER	DELIVER MESSAGE	CALL_ANSWERED	PUT_CALL
GTACK	RECEIVE_DRIVER	DELIVER ACK	CALL_DLVRED	RESPONSE_ACKED
GVCR	SEND_DRIVER	DISPATCH MESSAGE	SELECT_NEXT	SELECT_NEXT
GVACK	SEND_DRIVER	DISPATCH ACK	GIVE_ACK	GIVE_CALL_ACK
HOST_ DOWN		DAMAGE CONTAINMENT	HOST_FAILED	HOST_FAILED

Figure 2-5 Entry Points to the Call Handler Invoked by the Network Handler



NOTES:
1. EV I = EVENT I; EV I MUST PRECEDE EV I+1
2. CH INDICATES CALL HANDLER
3. NT INDICATES NET TRANCEIVER
4. DT, AK, NK = DATA, ACK AND NAK PACKERS
5. MS, MA = MESSAGE, MESSAGE ACK

Figure 2-6 . Interaction of Network Handler Components to Send Messages and Packets

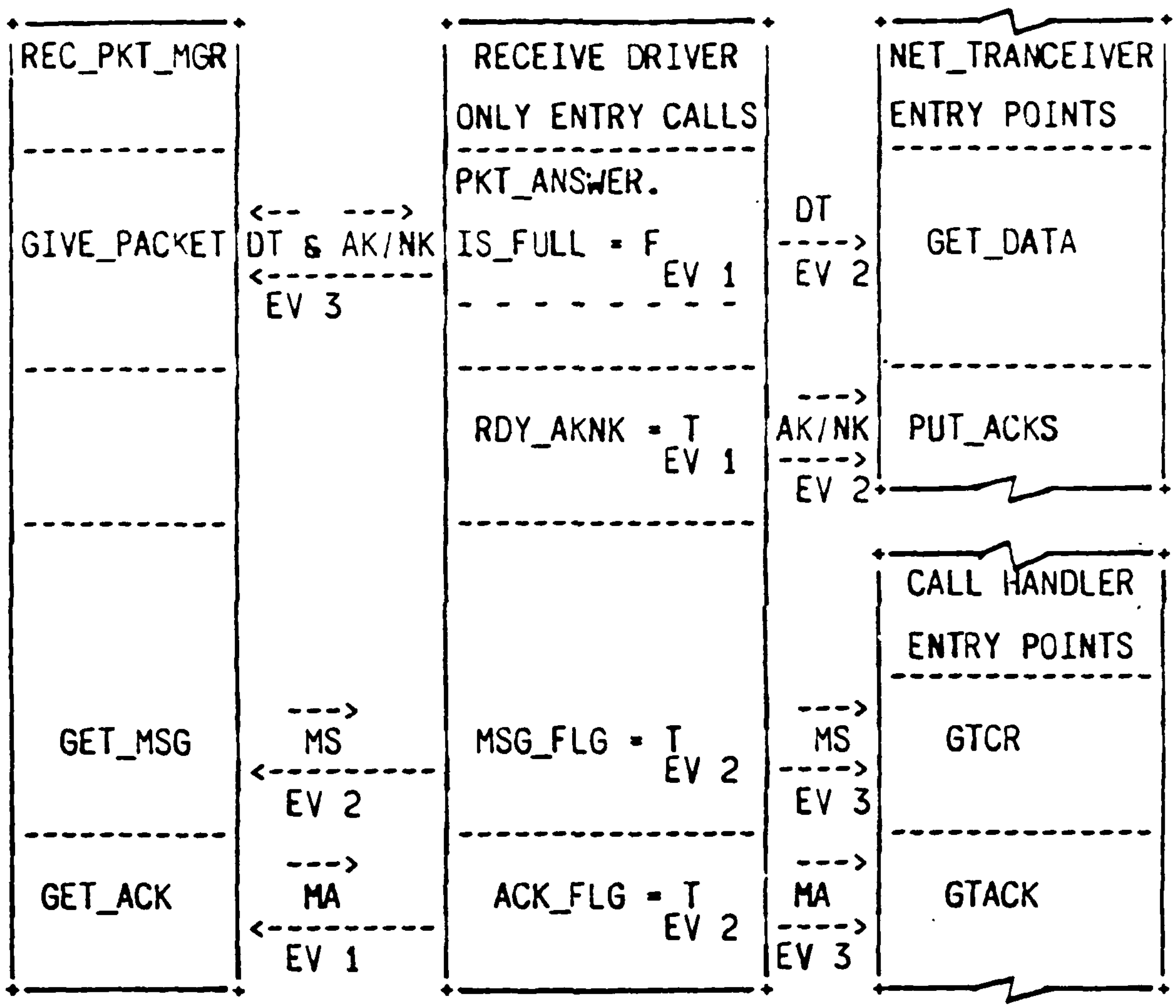


Figure 2-7 Interaction of Network Handler Components to Receive Messages and Packets

ENTRY	CALLER	PURPOSE	PACKET TYPE EXCHANGED
RDY_TO_SEND	NETWORK	READY FOR NEXT PACKET	ANY TYPE
RDY_TO_REC	NETWORK	PACKET ARRIVED	ANY TYPE
GET_DATA	RECEIVE_DRIVER	DELIVER DATA PACKET	DTP
PUT_ACKS	RECEIVE_DRIVER	GET ID OF ACCEPTED/ REJECTED PACKETS	--
SND_DATA	SEND_DRIVER	GET PACKET TO SEND	DTP, ADTP
GET_ACK	SEND_DRIVER	DELIVER ID OF AKED/ NAKED PACKETS	ADTP
GIVE_ACK	SEND_DRIVER	DELIVER ID OF PACKETS TO BE ACKED/NAKED	ADTP

Figure 2-8 Net Tranceiver Entry Points for RPC

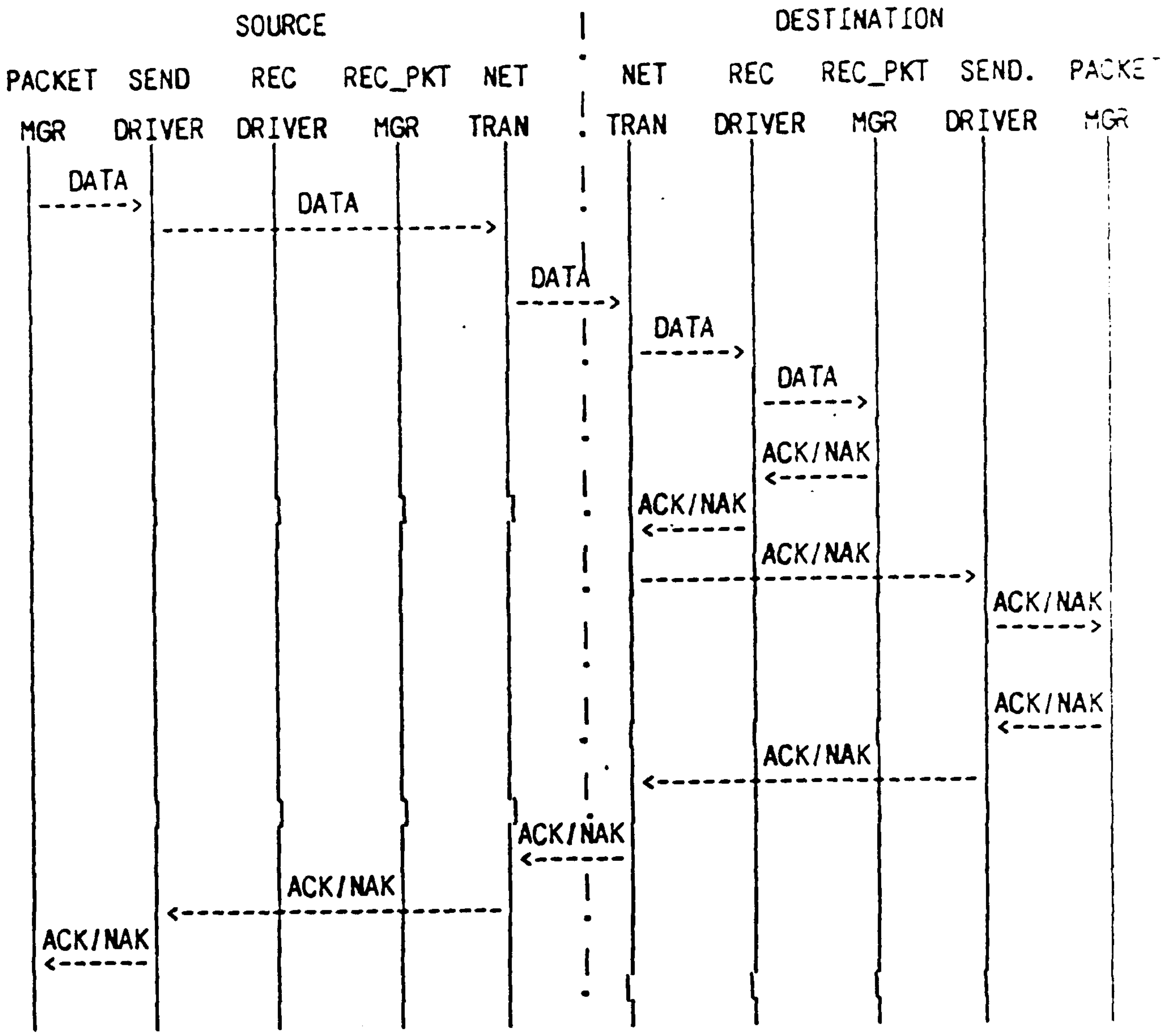
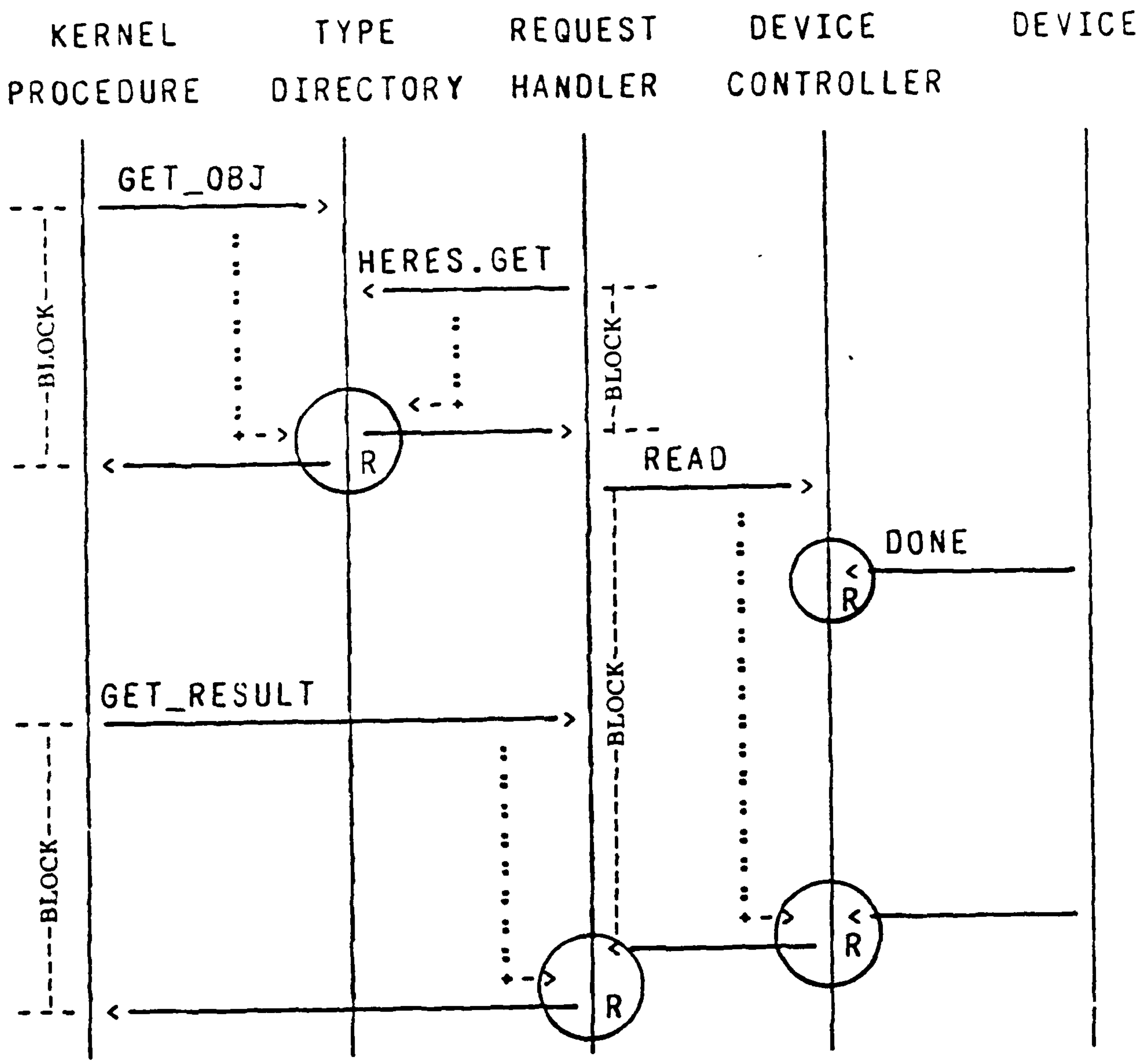


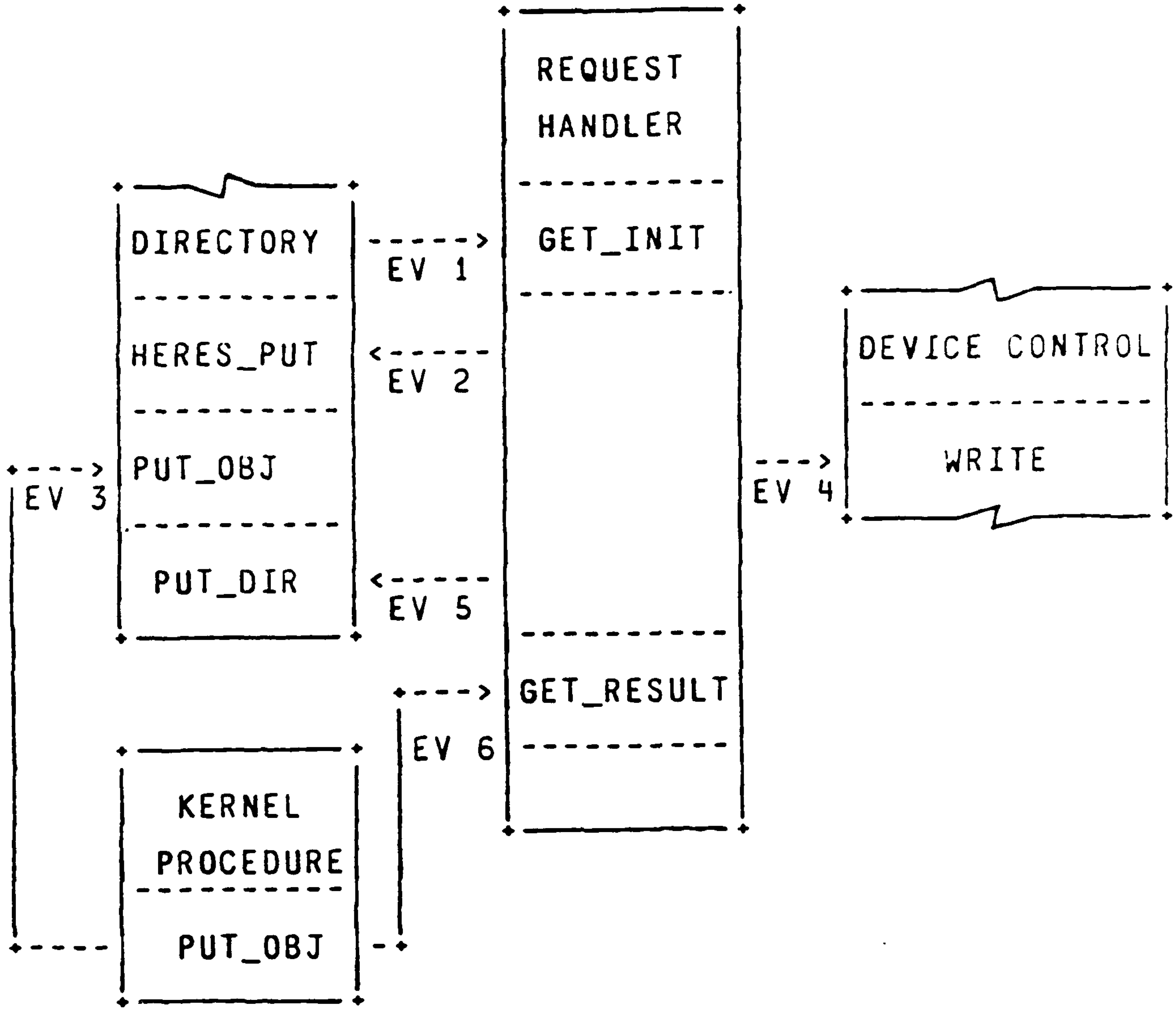
Figure 2-9 Timing Diagram of Data Packet and ACK/NAK Transmission



NOTE:

1. R = RENDEZVOUS FOR ENTRY CALLS WITH DASHED ARROWS.
2. FOR STABLE I/O ADD CAREFUL GET TO RIGHT OF REQUEST HANDLER.

Figure 2-10 Kernel Actions to Perform Object Storage and Retrieval



1. THE STRUCTURE AND SEQUENCE OF CALLS FOR ALL REQUEST HANDLERS IS SIMILAR.

2. THE REQUEST HANDLER'S SEQUENCE CAN BE EXPRESSED AS EV 1 (EV 2 EV 3 EV 4 EV 5 EV 6)*
   WHERE * = KLEENE STAR

Figure 2-11 Request Handler Sequence for Put Obj

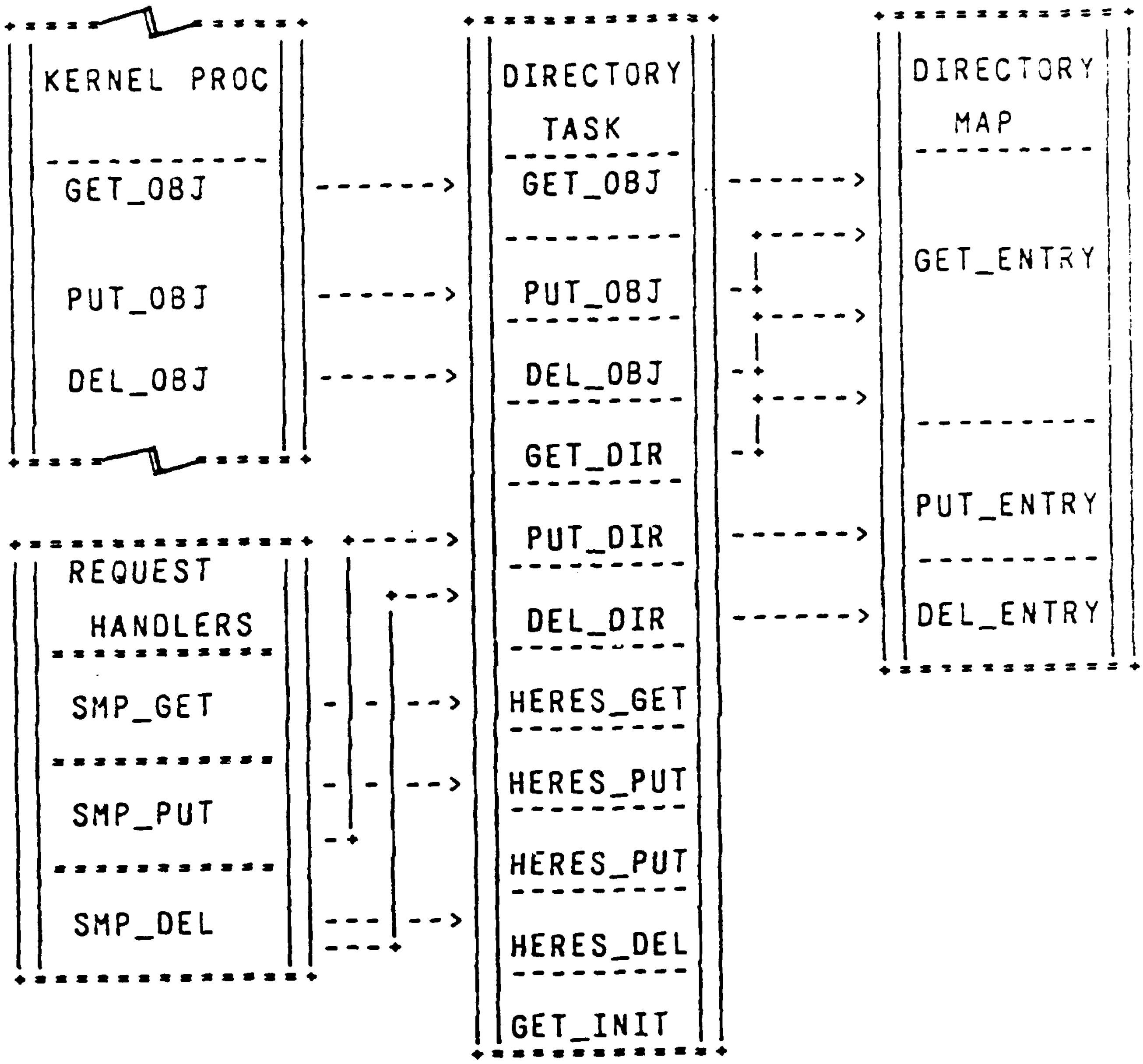
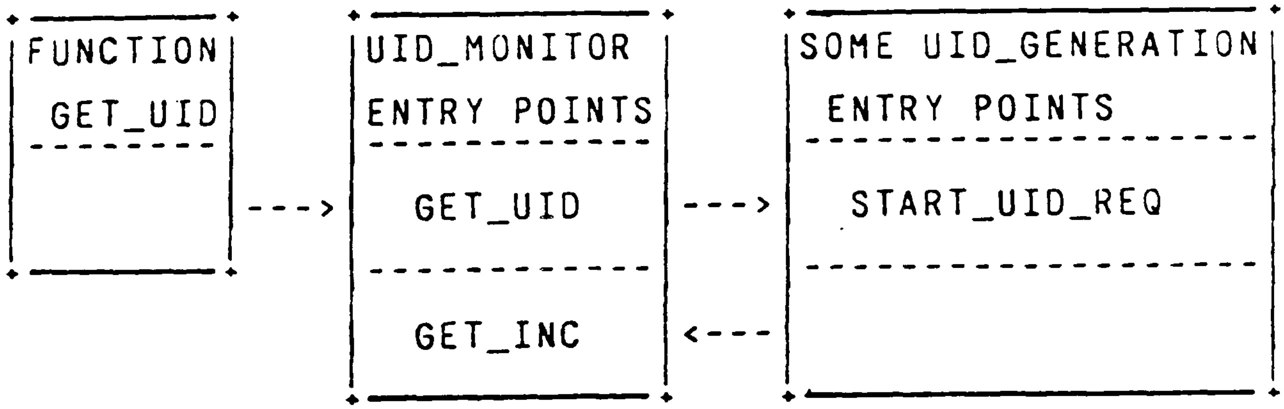


Figure 2-12 Directory Entries and Calls



INTERACTIONS

1. GET_UID CAUSES START_UID_REQ WHEN INCARNATION.

2. A START_UID_REQ IS ALWAYS RESPONDED TO BY A GET_INC.

3. A GET_INC MAY BE RECEIVED IF SOME OTHER HOST INCREMENTED THE INCARNATION NUMBER.

Figure 2-13 UID Monitor and UID Generation

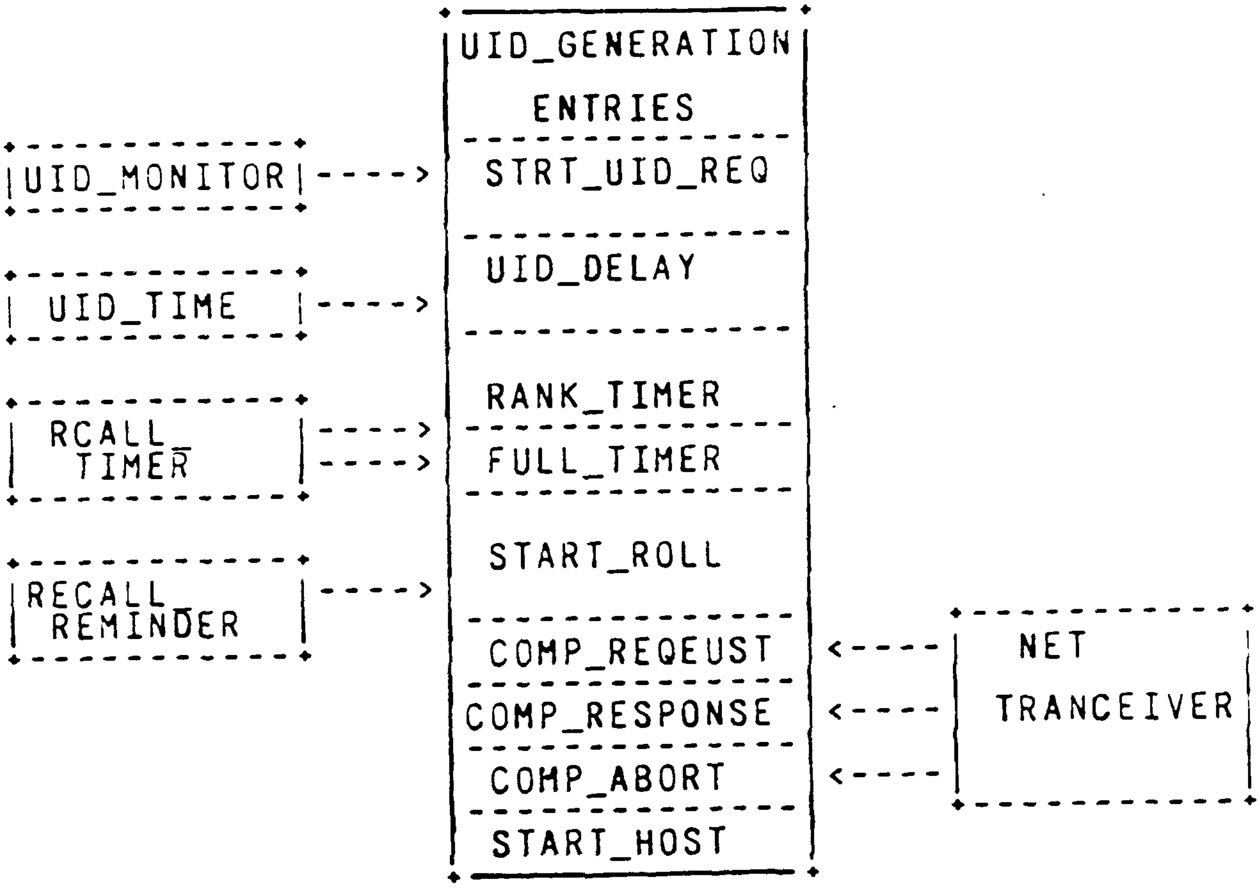


Figure 2-14 UID Generation Task Interactions

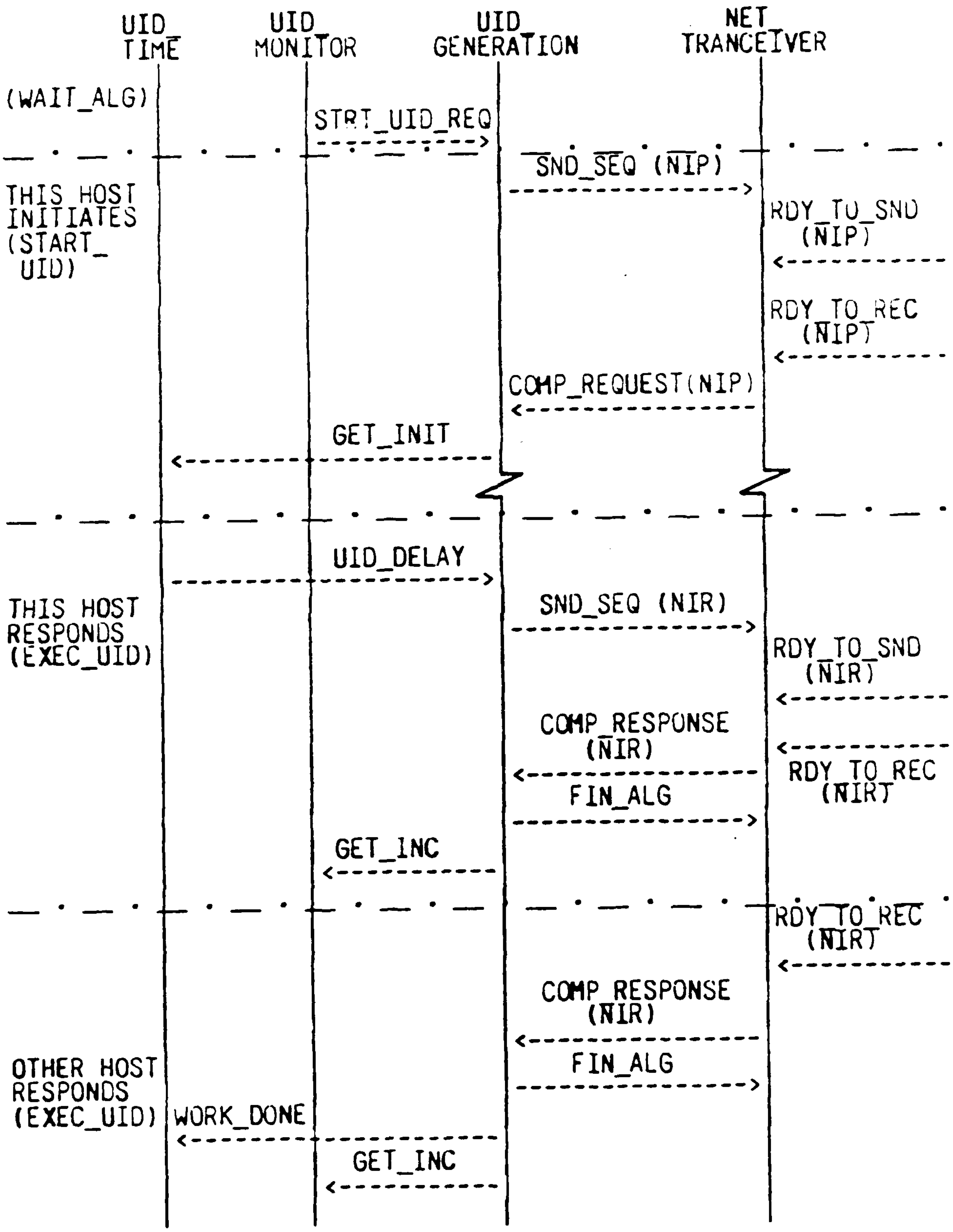


Figure 2-15 UID Generation Sequence

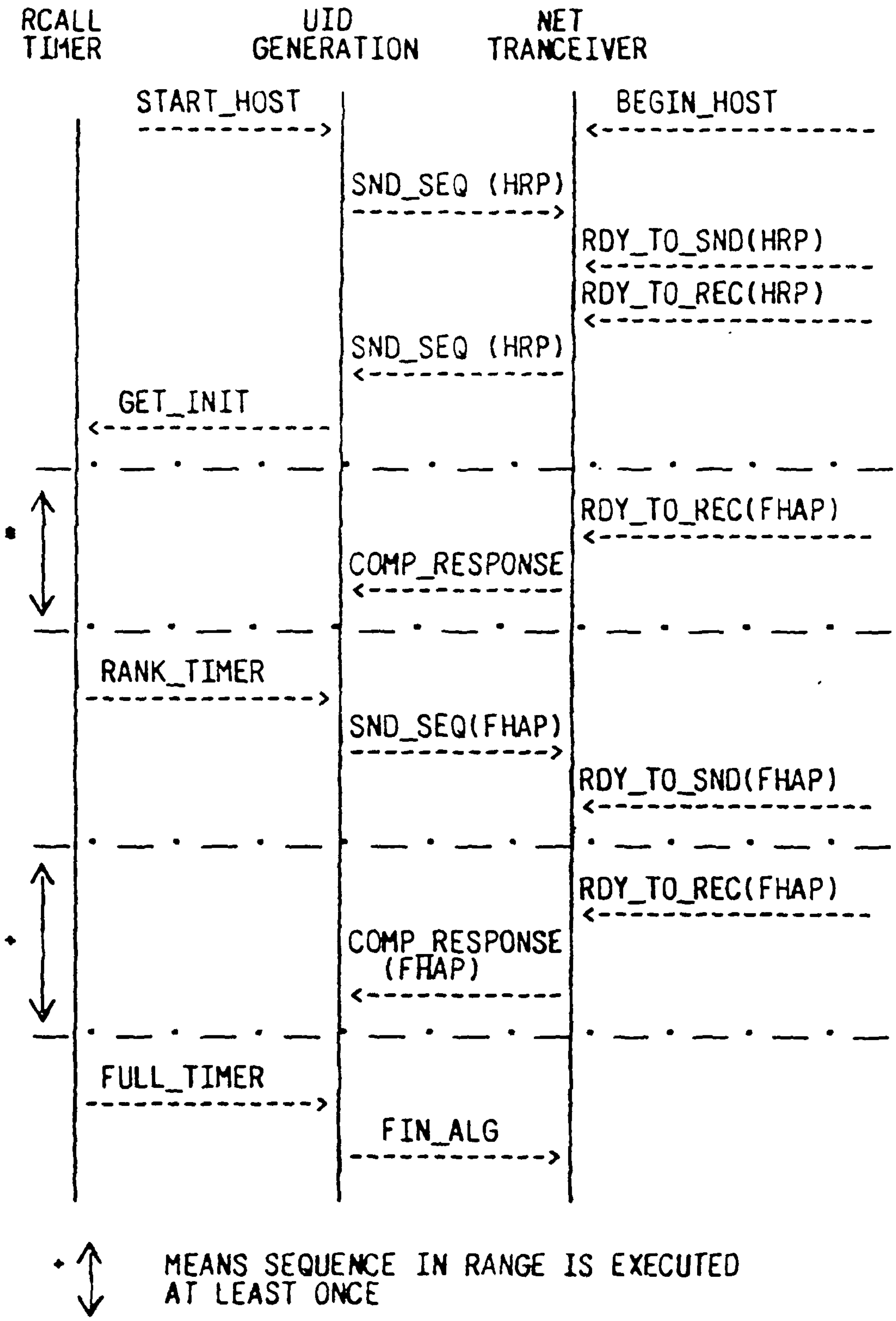


Figure 2-16 Host Restart Sequence

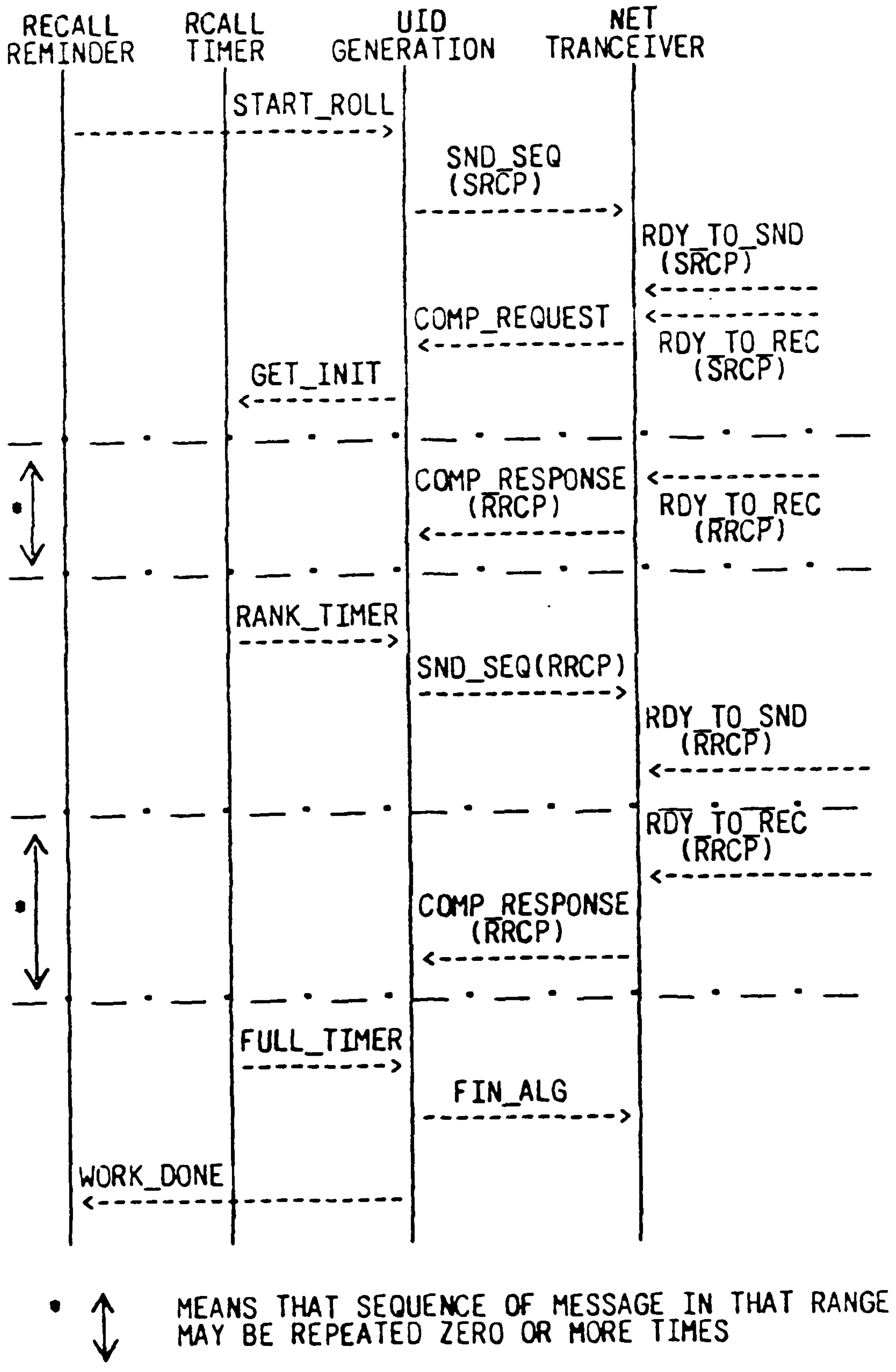


Figure 2-17 Roll Call Sequence

Chapter 3

PROCESS MANAGER DESIGN

This chapter presents a formal definition of the detailed designs of the Process/Transaction Manager in the Zeus distributed operating system, using CSDL.

Section 3.1, Machines Dictionary, describes the interfaces and behavior of various machines used in the design definiton of the Process Manager system. These machine defintions do not contain the details of the internal structure of the machines.Section 3.2 contains the type definitions for the various object types used in the entire design definition. For the sake of reader convenience this section has been divided into various sections depending on the type definitions related to some specific interfaces or sub-machines such as secondary storage, application processes, Operation Switch, database manager etc. Section 3.3 defines various procedures that are used by several machines in the design definitions. The details of the machine architectures are given in section 3.4 titled Realization Dictionary. These detailed design definitions include the architectures of the command processors used by the Process Manager machine. In each machine descripton, in addition to the CSDL definitions, we have also included informal algorithmic descriptions of the protocols used by that machine. These algorithmic descriptions are given in the BEHAVIOR parts of the machines and the procedures. Section 3.7, SYSTEM Process_Manager, defines the architecture of Process/Transaction Manager.

3.1 MACHINES DICTIONARY

3.1.1 Process Manager

```
SYSTEM Process_Manager

PUBLIC

            PM_TO_OS:  Small_Mailbox (OS_PM_Msg, PM_OS_Msg)
            PM_TO_MM :  Small_Mailbox(MM_PM_Msg,PM_MM_Msg)
            PM_TO_SS :  Small_Mailbox(SS_PM_Msg,PM_SS_Msg)
            PM_TO_UIDgen :  Small_Mailbox(UIDgen_PM_Msg,PM_UIDgen_
                                                Msg)
            CP_TO_SS:  LIST(Large_Mailbox(SS_PM_Msg,PM_SS_Msg))
            CP_TO_MM:  LIST(Large_Mailbox(MM_PM_Msg,PM_MM_Msg))

BEHAVIOR
```

Process_Manager machine contains several static machines as well as dynamic machines.The static machines, which are Router, Timer, MM_Port, SS_Port and PM_Database_Manager,are created at the same time as the PM_Manager whereas the dynamic machines, namely Processes and Command_Processors, are created in response to a request. PM_Manager also interfaces with the Stable Storage, Memory_Manager, Operation Switch, and UID_Generator.

END {Process Manager}

3.1.2 Router

```
Router IS

PUBLIC
        Router_TO_PM_CTRL_CMD:Small_Mailbox(PM_Router_CTRL_Msg ,
                                            Router_PM_CTRL_Msg )
        Router_TO_PM:Small_Mailbox(PM_Router_Msg ,
                                   Router_PM_Msg )
        Router_TO_Process:LIST(Large-Mailbox(Proc_Router_Msg,
                                             Proc_Router_Msg))
        Router_TO_OS : Small_Mailbox(OS_PM_Msg,PM_OS_Msg)
        Router_TO_CP:LIST(Small_Mailbox(PM_Router_Msg,
                                  Router_PM_Msg))
        Router_To_PMDB : Small_Mailbox(Invoke_Msg_Type,
                                             Invoke_Msg_Type)
```

BEHAVIOR

The Router machine is created when Process_Manager comes into existence.This machine provides communication path between submachines inside the PM and operation switch. Since Router is itself a submachine within PM ,it can not talk to any machine outside the PM directly; thus at Router's creation time the Router_To_OS mailbox is bound to PM_To_OS (a Public object of PM) through which Router will be able to communicate with OS.

END {Router}

3.1.3 Process

```
Process IS

    PUBLIC

        Process_To_PM:Small_Mailbox(Response_Msg,
                                    Appl_Req_Msg)
        Process_to_PM_PSCMD:Small_Mailbox(Response_Msg,
                                          Processor_Scheduler_Req_Msg)
        Process_TO_Router:Small_Mailbox(Router_Proc_Msg,
                                        Proc_Router_Msg )
```

BEHAVIOR

The Process Machine is created from the Process POOL in response to either an application request message or a request from a command processor. A Process at its creation time will be connected to the Router, through the Process_To_Router mailbox, and to the PM, through both Process_To_PM and Process_To_PM_ PSCMD. Process machine executes the program as per the Process_ Control_Block and also it executes the processor scheduler commands.
END {Process}

3.1.4 Timer

```
Timer IS

PUBLIC

        Timer_Command_Iface : Small_Mailbox(Timer_Command,
                                            Timer_Response)
        Interrupt : Event_Rec OUTLET

END     {Timer}
```

3.1.5 Delete Processor

```
Delete_Processor (My_Process_ID:Process_UID_Type,
                  Remote:BOOLEAN,
                  Requester:Process_UID_Type,
                  Work_Request:Appl_Req_Msg)

PUBLIC
     PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
     Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
     PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
     OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
     SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
     MM_Iface:Port(MM_PM_Msg,PM_MM_Msg)
     Descendent_Iface:LIST(Large_Mailbox(Response_Msg, Appl_Req_Msg)
```

BEHAVIOR

The Delete_Processor machine comes into existence in response to either an application request command 'Delete_Process' or a request from another command_processor. PM upon receiving a request for delete process creates this machine and passes the following parameters to the newly created machine:

My_Process_ID: UID for created machine.
Remote: Indicates whether the request is from a remote site or not.
Requester: UID of the process which invoked the operation.
Work_Request: The requested operation and its parameter. For
this machine the work request is: OP='Delete_Process'
Param=Process_UID_Type.

The Delete_Processor Machine to carry out the Delete operation executes one of the following procedures (based on the location of the process to be created).

```
Delete_Remote_Process  (Proc_ID: Process_UID_Type,
                        My_Process_ID: Process_UID_Type)
Delete_Local_Process   (Proc_UID: Process UID_Type)
```

Proc_UID is the UID of the process to be deleted.

END {Delete_Processor}

3.1.6 Create Processor

```
Create_Processor (My_Process_ID:Process_UID_Type,
                  Remote:BOOLEAN,
                  Requester:Process_UID_Type,
                  Work_Request:Appl_Req_Msg)
```

PUBLIC

```
PMDB_Iface:Port(PMDB_Response_Type, PMDB_Request_Type)
Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
OS_Iface:Small_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
MM_Iface:Port(MM_PM_Msg,PM_MM_Msg)
Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
UID_Gen_Iface:Port(UIDgen_PM_Msg, PM_UIDgen_Msg)
```

BEHAVIOR

The create_processor machine is created in response to either the application request command 'Create Process' or a request from another Command_Processor.

Following Parameters will be passed to the Create_Processor at the time of creation:

My_Process_ID: UID of the created command processor machine
Remote: Indicates whether the request is from a remote site or not.
Requester: UID of the process which invoked the operation
Work_request: The requested operation and its parameter.
For this machine the Work_Request is:
OP = 'Create_Process'
Params=Program_UID, Data_UID, host_ID, Time_Out

Following procedures are executed on Create_Processor machine:

```
Create_Remote_Process (Param: Create_Param,
                       My_Process_ID: Host_ID_Type)
Create_Local_Process (Params: Create_Param)
```

Create_Param is the parameters needed to create the new process.

END {Create_Process}

3.1.7 PM Database Manager

```
PM_Database_Manager

PUBLIC
        SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
        Database_Port:Small_mailbox(Port_Msg(PMDB_Request_Type),
                  Port_Msg(PMDB_Response_Type))
        OS_Iface :  Small_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
```

BEHAVIOR

The PM_Database_Manager machine comes into existence when Process_Manager is created. This machine contains a set of objects from which information about the active processes can be acquired. Also, this information is essential for PM to carry out its functions reliably.

END {PM_Database_Manager}

3.1.8 Port Multiplexer

```
Port_Multiplexer( T1 : TYPE , T2 : TYPE)

PUBLIC
     Device_Iface : Small_Mailbox(Port_Msg(T1),Port_Msg(T2))
     Iface : Small_Mailbox(Port_Msg(T2),Port_Msg(T1))
```

BEHAVIOR

The Port_Multiplexer machines are created when Process_Manager comes into existence. These machines provide communication path between submachines inside the PM and independent machines outside the PM namely Stable_Storage, Primary_Memory and UID_Generation. The proper message types for the communication ports must be passed to these machines at their creation time.

END {Port_Multiplexer}

3.1.9 End_Transaction Processor

```
End_Trans_Processor(My_Process_ID:Process_UID_Type,
                    ERP :  BOOLEAN,
                    Requester:Process_UID_Type,
                    Work_Request:Appl_Req_Msg)
```

PUBLIC

```
PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
```

BEHAVIOR

The End_Tran_Processor machine is created in response to either an applciation request command 'End_Transaction' or a request from another command processor. Following parameters must be passed to the End_Trans_Processor at the time of creation:

My_Process_ID: UID of created machine.
ERP : Indicates whether the requester wants to establish a recovery point or not.
Requester: UID of the process which invoked the operation or not.
Work_Request: The requested operation and its parameters. For this machine the work request is:
OP='End_Transaction'
Param=NULL.

The End_Transaction operation is the commit point for the Transaction. Thus, the Commit procedure is executed to carry out this operation. The END_Transaction operation is the commit point for the transaction. Thus, for an outermost transaction, execution of the End_Transaction statement means permanence of all updates made within this transaction.

END {End_Trans_Processor}

3.1.10 Abort Processor

```
Abort_Processor(My_Process_ID:Process_UID_Type,
                Remote:BOOLEAN,
                Requester:Process_UID_Type,
                TUID:Transaction_UID_Type)
```

PUBLIC

```
PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
```

BEHAVIOR

The Abort_Processor machine comes into existence in response to either the application request command 'Abort' or a request from another command processor. Following param are passed to this machine at its creation time:

My_Process_ID: UID of created machine
Remote: Indicates whether the request is from a remote site or not.
Requester: UID of process which invoked the operation.
Work_Request: The requested operation and its parameter. For this machine the work request is
OP = 'Abort'
Param = NULL or Process_UID.

The Abort operation for Transactions terminates the execution of the current block and restores the state of the local variables and global objects to their values before the beginning of transaction. If Abort command is used within a process it terminates the process. Procedure Abort in this machine perform Abort operation.

END {Abort_Processor}

3.1.11 Commit Processor

```
Commit_Processor(My_Process_ID:Process_UID_Type,
                 Remote:BOOLEAN,
                 Requester:Process_UID_Type,
                 TUID:  Transaction_UID_Type)
```

PUBLIC

```
PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
```

BEHAVIOR The Commit_Processor machine is created in response to either the application request command 'Commit' or a request from another command processor. Following are the parameters which must be passed to this machine at its creation time.

My_Process_ID: UID of created machine.
Remote: Indicates whether the request is from a remote site or not.
Requester: The UID for the process which invoke the operation.
TUID: The UID of transaction to be committed.

The Commit_Processor executes the Commit_Protocol_Terminator to carry out the commit operation. The Commit_Transaction operation makes all the updates which have been performed by a transaction permanent. This function can be called only by a non-transaction process that has created some concurrent transaction (by executing the Create_Transaction function). Therefore, no nested transaction is committed by calling this function; the commitment of a nested transaction occurs when its parent transaction executes its End_Transaction command. The execution of this command for a nested transaction is still valid; however, such an invocation of this command will not commit a nested transaction.

END {Commit_Processor}

3.1.12 Rollback Processor

```
Rollback_Processor(My_Process_ID:Process_UID_Type,
                   TUID:  Transaction_UID_Type,
                   RP_Number:INTEGER)
```

PUBLIC

```
      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
```

BEHAVIOR

The Rollback_Processor machine is created in response to either the application request command 'Rollback' or a request from another command processor. Following are the parameters which must be passed to this machine at its creation time.

My_Process_ID: UID of created machine.
TUID: The UID of transaction to be rolled back.
RP_Num : the recovery point number where the process or transaction is to be rolled back to.

The Rollback operation within a process restores the state of all the local objects to their values which they possessed at the time the RP_num was established. The changes on global objects, which have been made by transactions within that process, remain permanent if the transactions performing those are committed.

The Rollback operation within a transaction restores the state of all the local and global objects to their values that they had at the time the RP_num was established.

END {Rollback_Processor}

3.1.13 ERP Processor

```
ERP_Processor (My_Process_ID:Process_UID_Type,
               Requester :  Process_UID_Type,
               Work_Request:Appl_Req_Msg)
```

PUBLIC

```
      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
```

```
OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
MM_Iface:Port(MM_PM_Msg,PM_MM_Msg)
```

BEHAVIOR

The ERP_Processor machine is created in response to either the application request command 'Establish_Recovery_Point' or a request from another command processor. Following are the parameters which must be passed to this machine at its creation time.

My_Process_ID: UID of created machine.
Requester: The UID for the process which invoke the operation.
Work_Request : The requested operation & its parameters.

The ERP operation saves the current state of the process or Transaction object in stable storage. Successive calls to this function increments the RP_Num by one. The updated RP_Num indicates the latest recovery point number within the ontext in which it is called. the first recovery point for each process or transaction has the value of zero and it is established automatically hen a process or transaction starts its execution.

END {ERP_Processor}

3.1.14 DRP Processor

```
DRP_Processor (My_Process_ID:Process_UID_Type,
               Remote : BOOLEAN'
               Requester : Process_UID_Type,
               Work_Request:Appl_Req_Msg)
```

PUBLIC

```
PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
```

BEHAVIOR

The DRP_Processor machine is created in response to either the application request command 'Discard_Recovery_Point' or a request from another command processor. Following are the parameters which must be passed to this machine at its creation time.

My_Process_ID: UID of created machine.
Remote: Indicates whether the request is from a remote site or not.
Requester: The UID for the process which invoke the operation.
TUID: The UID of transaction to be committed.

The DRP operation discards all recovery points the whose RP_num is equal to and includes specified recovery points between.

```
END   {DRP_Processor}


END        {Machine Dictionary}
```

3.2 TYPES DICTIONARY

3.2.1 UID_Type Definition

```
        Host_ID_Type IS (0..((2**10)-1))

        Unique_Number_Type IS (Large_step_No:(0..((2**22)-1)),
                               Seq_No       :(0..((2**32)-1)),
                               Host_ID      :(0..((2**10)-1))
                              )

        Type_Name IS Unique_Number_Type

        UID_Type IS (Object_ID:Unique_Number_Type,
                     Type_ID: Unique_Number_Type
                    )

       Extended_UID_Type IS (UID       : UID_Type,
                             Host_Hint : Host_ID_Type)

       Type_Type_Manager_UID_Type IS
       MODEL UID_Type
       LET TTM-UID: Type_Type_Manager_UID_Type
       INVARIANT
           TTM_UID.Object_ID.Large_Step_No := 1  AND
           TTM_UID.Object_ID.Seq_No := 0  AND
           TTM_UID.Object_ID.Host_ID := 0  AND
           TTM_UID.Type_nique_No.Large_Step_No := 0  AND
           TTM_UID.Type_nique_No.Seq_No := 0
       END {Type_Type_Manager_UID_Type}

       Process_Manager_UID_Type IS
       MODEL UID_Type
       LET PM_UID:  Process_Manager_UID_Type
       INVARIANT
           PM_UID.Object_ID.Large_Step_No := 1 AND
           PM_UID.Object_ID.Seq_No := 0  AND
           PM_UID.Object_ID.Host_ID := 0  AND
           PM_UID.Type_Unique_No.Large_Step_No := 0 AND
           PM_UID.Type_Unique_No.Seq_No := 1
       END {Process_Manager_UID_Type}
```

```
Process_UID_Type IS
MODEL Extended_UID_Type
LET Proc_UID:Process_UID_Type
INVARIANT
     Proc_UID.UID.Unique_Number_Type.Large_Step_No=0  AND
    (Proc_UID.UID.Type_Unique_No.Seq_No=1  OR
     Proc_UID.UID.Type_Unique_No.Seq_No=2)

END {Process_UID_Type}

Transaction_UID_Type IS
MODEL Extended_UID_Type
LET TR_UID:Transaction_UID_Type
INVARIANT
     Trans_UID.UID.Type_Unique_No.Large_Step_No=0  AND
     Trans_UID.UID.Type_Unique_No.Seq_No=2

END {Transaction_UID_Type}
```

3.2.2 Type definitions for Process Manager's database

```
Register IS INTEGER

Base_Bound IS (Base,Bound:Register)

Set_of_Rights IS (Owner,Abort,Suspend,Update_Priority,Restart,
                  Start,Terminate,Destroy)

Process_Status_Type IS (Non-Existent,Running,Aborted,Suspended,
                        Completed,Crashed)

Transaction_status_Type IS (Non_Existent,Uncommitted,Commit-
                            Pending,Committed,Completed,Crashed,
                            Aborted)

Process_Access_Record IS (User_ID:UID_Type,
                          Access_Rights:Set_of_Rights
                         )

Process_Record IS   (PUID:Process_UID_Type,
                     Access_Control_List:Process_Access_Record,
                     Priority:INTEGER,
                     Proc_State:Process_Status_Type,
                     Time-Out:INTEGER
                     LRP:INTEGER,
                     Creation_Time:INTEGER
                    )

Transaction_Record IS
```

```
                        (PUID:Transaction_UID_Type,
                         Access_Control_List:Process_Access_Record,
                         Priority:INTEGER,
                         Proc_State:Process_Status_Type,
                         Time_Out:INTEGER,
                         Trans_State:Transaction_Status_Type,
                         LRP : INTEGER,
                         Creation_Time:INTEGER
                        )

Active_Process_Record IS

                        [Transaction:Transaction_Record[]
                         Process:Process-Record
                        ]UNION

Process_Control_Block IS

                        (PUID:UID_Type,
                         Process_Status_Registers:Register ARRAY,
                         Process_Base_Bound:Base_bound,
                         Data_Base_Bound:Base_Bound
                         RP_Num:INTEGER
                        )

Descendent_Record IS (Child_ID:Process_UID_Type,
                      Grand_Children:Process_UID_Type ARRAY,
                      Indirectly_Modified_Obj:Extended_UID ARRAY)

RP_Child_Relation IS (RP_Number:INTEGER,
                      Desendent_Table_Indx:INTEGER
                      {This index indicates that all transactions
                      and processes that were created after
                      the recovery point RP_Number are stored
                      in the Descendent_Table starting with index
                      Descendent_Table_Indx+1}
                      )

Parent_Child_Info IS

                        (Process_ID:UID_Type,
                         Parent_UID:UID_Type,
                         Root_UID:Transaction_UID,
                         Top_Level:BOOLEAN {True if the process specified
                                   by the Proc_ID field is a transaction
                                   whose parent is a non-transaction
                                   process}
                         Map_Field:(Sequential, Concurrent) {with respect
                                    the parent process/transaction}
                         Location:(Local , Remote)
                         Descendent_Table:Descendent_Record ARRAY,
                         RP_Child_Map:RP_Child_Relation ARRAY
                         )
```

```
Directly_Modified_Objects IS
              (PUID:Process_UID_Type,
               Modified_Objects:Extended_UID_Type ARRAY
              )

{NULL is of Type ARRAY with no element in it}
NULL IS INTEGER ARRAY

RP_Num IS INTEGER

{RP_Number is an array which may have 0,1 or more element}
RP_Number IS RP-Num ARRAY

Operation_Type IS [ERP , (OP :(Establish_Recovery_Point,
                               End_Transaction)
                          Param : NULL
                         )                        []
                   Commit_Transaction , UID_Type  []
                   Rollback , RP_Number] UNION

Current_Operation_Info IS

                    (PUID:Process_UID_Type,
                     OP:Operation_Type
                    )


Time_Stamp IS INTEGER

RP_Data_Record IS (local_var:ABSTRACT,
                   PCB:Process_Control_Block,
                   TS:Time_Stamp)

Memory_Allocation IS (Starting_Addr:INTEGER,Length: INTEGER)
```

3.2.3 PM To Process Interface

```
PUID IS UID_Type
DUID IS UID_Type
P IS [Program_UID, PUID []
      Data_UID, DUID []
      Host_ID, Host_ID_Type[]
      Expected_Time, INTEGER] UNION
Create_Param is P ARRAY

{Suspend_Param is defined as an array which may contain up to 2 elements}
 S IS [Process_UID, Process_UID_Type []
       Delay_Time, INTEGER ] UNION
```

```
{DRP_Param is an array which may contain up to 3 elements}
 DRP IS [Process_UID : Process_UID_Type []
         RPnums,RP_Number] UNION

 Appl_Req_Operations IS (Create_Process,Delete_Process,
                         Process_Status,Suspend,
                         Resume,Establish_Recovery_Point,
                         Discard_Recovery_Point,Rollback,
                         Last_Recovery_Point,Begin_Transaction,
                         End_Transaction,Create_Transaction,
                         Transaction_Status,Commit_transaction,
                         Abort
                        )

{T1 operations are: Create_Process, Create_Transaction.}
 T1 IS (OP : Appl_Req_Operations,
        Param : Create_Param)

{T2 operations are: Delete_Process, Restart, Last_Recovery_Point,
 Abort.}
 T2 IS (OP : Appl_Req_Operations,)
        Param : UID_Type ARRAY)

{T3 operations are: Process_Status, Resume, Transaction_Status,
 Commit_Transaction.}
 T3 IS (OP : Appl_Req_Operations,
        Param : UID_Type)

{T4 operations are: Establish_Recovery_Point, Begin_Transaction,
 End_Transaction & all other operations whose responses are NULL.}
 T4 IS (OP : Appl_Req_Operations,
        Param : NULL)

{T5 operations are: Begin_Transaction, Create_Transaction.}
 T5 IS (OP : Appl_Req_Operations ,
        Param : Transaction_UID_Type)

{T6 operations are: Last_Recovery_Point, Establish_Recovery_Point.}
 T6 IS (OP : Appl_Req_Operations,
         Param : RP_Num)

{Message Type for Application functions}
 Appl_Req_Msg  IS  [Create , T1 []
                    Array_UID_Param , T2 []
                    UID_Param , T3 []
                    NULL_Param , T4 []
                    Suspend , S ARRAY []
                    Discard_Recovery_Point , DRP ARRAY []
                    Rollback , RP_Number ] UNION

 {Application commands response from PM to process}

 ACK IS (Failure , Success)
```

```
Appl_Resp_Msg    IS    [NULL_Resp , T4 []
                        Tr_UID_Resp , T5 []
                        RP_Num_Resp , T6 []
                        Create_Process , Process_UID_Type []
                        Process_Status , Process_Status_Type []
                        Transaction_Status , Transaction_Status_type] UNION

{Message request Type for Processor Scheduler Commands}

 Processor_Scheduler_Cmds IS (Get_PCB, Load_PCB
                                  Run_Process,Stop_Process)

 T8 IS (OP : Processor_Scheduler_Cmds,
         Param : NULL
        )
 T9 IS (OP : Processor_Scheduler_Cmds,
         Param : PCB_Type
        )

 Processor_Scheduler_REQ_Msg  IS  [NULL_Param , T8 []
                                    PCB_Param , T9] UNION

{PS Commands response Type from PM to Process}
 Processor_Scheduler_Resp_Msg    IS  [NULL_Resp , T8 []
                                       PCB_Resp , T9 ] UNION

 Error_Condition_Code    IS    (Non_Existent_Transaction,
                                Access_Control_Violation,
                                Non_Existent_Process,
                                Host_Inaccessible,
                                Time_Out,
                                Undefined_Error
                                Illegal_Command)


 PM_Proc_Resp_Msg        IS    [Appl,Appl_Resp_Msg[]
                                PS,Processor_Scheduler_Resp_Msg] UNION

Response_Msg       IS    [Success,PM_Proc_Resp_Msg[]
                          Failure,Error_Condition_Code] UNION

 Search_Rec IS [Success,TYPE[]
               Fail, NULL] UNION


 Suspended_Caller_Rec IS (UID : Process_UID_Type,
                          Operation : Appl_Req_Msg)

 Machine_Index_Type IS [Application, Process INDEX[]
                        Delete, Delete_Processor INDEX[]
                        Commit, Commit_Processor INDEX[]
                        Create, Create_Processor INDEX[]
```

```
                                Rollback, Rollback_Processor INDEX[]
                                ERP, ERP_Processor INDEX[]
                                Abort, Abort_Processor INDEX[]
                                End , End_Trans_Processor] UNION
```

3.2.4 PM Database Interface Types

```
All_Rec_Type IS  [APR,Active_Process_Record[]
                  PCI,Parent_Child_Info[]
                  LMO,List_of_Modified_Object[]
                  COI,Current_Operation_Info] UNION

Add_Modify_Param IS (OP : (Add , Modify),
                     Param : All_Rec_Type ARRAY)

PMDB_Modify_Operations IS [Add_Modify , Add_Modify_Param[]
                           Delete , UID_Type] UNION

Database_Query_Type IS (Process_Status_Record,
                        Modified_Object,
                        Parent_Child_Info,
                        Get_UID)

PMDB_Request_Type IS [Update, PMDB_Modify_Operations[]
                      Query, (UID:Process_UID_Type,
                              Request:Database_Query_Type)] UNION

PMDB_Response_Type IS [Update, ACK []
                       Query, All_Rec_Type] UNION
```

3.2.5 PM to SS Interface To OS Interface

```
{Request_Message type definition for request message from
process Manager to Stable Storage.}


RP_Label_Type IS  (UID: UID_Type,
                   RPnum: RP_Num)

Address : INTEGER
RW_Param  IS  (Label: RP_Label_Type,
               Starting_Addr: Address,
               Len: INTEGER)

PMDB_Log_Buffer:  LIST(PMDB_Modify_Operations)

Append_Param IS (List_of_Rec:  LIST(TYPE),
                 Stable_Storage_Filename:  CHAR ARRAY)

{The Append operation will append the given list of record
 to the specified file on stable storage which has to be
```

the file of same record type. The special case is appending PMDB_Log_Buffer to the Differential file on stable storage.}

```
{Message type definition for Process Manager & Stable Storage
 Communication.}

PM_SS_Msg IS  [Read_Write, (OP : (Read, Write),
                            Param : RW_Param) []
               Append, Append_Param []
               Find_Length, RP_Label_Type ] UNION

{Response Message type definition for Stable Storage to Process
 Manager}

SS_PM_Msg IS [ACK_Resp , (OP : (Read, Write, Append),
                          Param : ACK) []
              Find_Length,INTEGER] UNION
```

3.2.6 PM TO UIDgen Interface

```
{Message Type definition for PM & UID generator communication.}

 PM_UIDgen_Msg IS Type_Name

{Response Message Type definition for PM & UID generator
 Communication.}

 UIDgen_PM_Msg IS UID_Type
```

3.2.7 PM TO MM Interface

```
{Type definition for Process Manager &
Main Memory Communication}

PM_MM_Msg:ABSTRACT {Request}
MM_PM_Msg:ABSTRACT {Response}
```

3.2.8 PM TO OS Interface

```
{type definition for Process Manager
& Operation Switch Communication}

Other_Msg_Type:  ABSTRACT
General_Msg_Type IS  [Appl_Cmds,Appl_Req_Msg[]
                      Response,Response_Msg[]
                      Other_Msg,Other_Msg_Type] UNION
```

```
Invoke_Msg_Type  IS  (Sender : Extended_UID_Type,
                      Operation : General_Msg_Type,
                      Reciever : Extended_UID_Type,
                     )

PM_OS_Msg IS Invoke_Msg_Type
OS_PM_Msg IS Invoke_Msg_Type
```

3.2.9 PM TO Router Interface

```
Router_PM_Msg IS Invoke_Msg_Type
PM_Router_Msg IS Invoke_Msg_Type

Router_PM_CTRL_Msg IS ACK
PM_Router_CTRL_Msg IS (OP : (Create_Mbx, Delete_Mbx,
                             Create_CP_Mbx, Delete_CP_Mbx)
                             Param : Process_UID_Type)
```

3.2.10 PM TO Timer Interface

```
{Type definition for PM To Timer communication.}

Event_Rec IS (Caller_UID : Process_UID_Type,
              Operation  : Appl_Req_Msg,
              Action     : [Retry , INTEGER[]
                            Time_Out, Null] UNION
              Time_Delay : INTEGER
             )

Timer_Command IS [Set , Event_Rec[]
                  Clear, Process_UID] UNION

Timer_Response  IS  ACK
```

3.2.11 Router TO Process Interface

```
Router_Proc_Msg  IS  ABSTRACT

Proc_Router_Msg  IS  ABSTRACT
```

3.2.12 Command Processor Interface Types

```
Command_Machine_Type IS (Delete, Create, Commit, Abort,End_Trans,DRP,
                         Rollback, ERP)

Action_Type IS (New_Process, Delete_Process,Run_Process,.......)

Action_Param_Type IS [tag1, PCB_Type []
                      tag2, UID_Type] UNION
```

```
CP_To_Controller IS [Create, (Machine_Type : Command_Machine_Type,
                              Descendent_Mbx_Indx : INTEGER,
                              Caller_Type:Command_Machine_Type,
                              Work_Request : Appl_Req_Msg) []
                     destroy,  Command_Machine_Type []
                     Service_Call,  (Action: Action_Type,
                                     Param : Action_Param_Type)] UNION

Controller_To_CP IS [t1, ACK []
                     t2, UID_Type] UNION
```

3.2.13 Definition of abstract data type for List manipulation

```
LIST (T:TYPE) IS
MODEL [T ARRAY]
Let %L:LIST (T)(%L)
{T must have a field containing UID.}

INVARIANT
INIT        %L.dom = 0

OFUN Add(V:T)
PRE TRUE
POST %L'.Hib = %L.Hib +1
           AND %L'.High = v

OFUN Delete (V:UID_Type) RETURNS BOOLEAN
PRE TRUE
POST %L.Hib = %L.Hib-1 AND Delete'  = TRUE
     IFF  exists
     IF not ( i (v=%L[i].UID)) Then
      Delete' = FALSE


VFUN Search (V:UID_type)_RETURNS Search_Rec
PRE TRUE
POST IF (There exist i:%L.lob<i<%L.hib AND (v=%L[i].UID)) Then
     Search_Rec.FLag = TRUE
     Search_Rec.x.value = %L[i]
     Search_Rec.x.tag = Success

     and

     IF not (There exist i:%L.Lob<i<%L.hib (v=%L[i].UID)) Then
     Search_Rec.FLAG = FALSE
     Search_Rec.x.tag = Fail

VFUN Is_Empty RETURNS BOOLEAN
PRE TRUE
POST %L.dom>o=> Is_Empty = TRUE
```

```
     NOT (%L.DOM>0_=> Is_Empty = FALSE
BEHAVIOR
     {This function indicates whether the list is
      empty or not.}


VFUN Hi_Bound RETURNS INTEGER
PRE TRUE
POST Hi_Bound' = %L.hib
BEHAVIOR
     {This function returns the largest index of
      the list.}


VFUN Element(i:INTEGER) RETURNS T
PRE TRUE
POST Element:=%L(i)
BEHAVIOR

  {This function RETURNS the element of LIST
   LIST referenced by index i.}

VFUN Get_List_Indx}i(Proc_UID:Process_UID_Type) RETURNS INTEGER
     PRE TRUE
     POST %L[Get_List_Indx(Proc_UID)].UID=Proc_UID
     BEHAVIOR
          {This function returns the index of element
           whose id number is Proc_UID.}


OFUN Hi_Extend RETURNS INTEGER
     PRE TRUE
     POST %L'.hib = %L.hib + 1
```

END {LIST}

3.2.14 Definition of abstract data type for Small_Mailbox.

```
ACTIVE Small_Mailbox (T1:TYPE,T2:TYPE) IS
MODEL [Request:T1 INLET,Response: T2 OUTLET]
LET %M1,%M2: Small_Mailbox
        %M3: Large_Mailbox
        %P : Port

COMPLEMENTS %M1.Request,%M2.Response;
            %M2.Request,%M1.Response;
            %M1.Request,%M3.Response;
            %M3.Request,%M1.Response;
            %M1.Request,%P.Out;
            %M1.Response,%P.In
```

```
OFUN Get RETURNS T1
     PRE TRUE
     POST Get' = %M1.Request.Window &
                 %M1.Request.Flag = TRUE

OFUN Put (x:T2)
     PRE TRUE
     POST %M1.Response.Window = x &
          %M1.Response.Flag = FALSE

VFUN Went RETURNS BOOLEAN
     PRE TRUE
     POST Went' = %M1.Response.Flag

VFUN Came RETURNS BOOLEAN
     PRE TRUE
     POST Came' =  %M1.Request.Flag

        VFUN Send (Msg:T2)RETURNS BOOLEAN

     PRE %M1.Response.Flag=TRUE
            {to make sure previous message has been read.}

     POST Send=TRUE IFF %M1.Response.Window=Msg &
                        %M1.Response.Flag=TRUE

     BEHAVIOR

     %M1.Response.Window::=Msg
     %M1.Response.Flag::=FALSE
     WHEN
         %M1.Response.Flag --> Send::=TRUE

     END

END {Small_Mailbox}
```

3.2.15 Definition of abstract data type for Large_Mailbox

```
ACTIVE Large_Mailbox (T1:Type, T2:Type) IS
MODEL [Request:T1 INLET, Response:T2 OUTLET,
       UID: Process_UID_type,Indx:Machine_Index_Type]

LET %M1,%M2: Large_Mailbox,
        %M3: Small_Mailbox,
        %P : Port
```

```
COMPLEMENTS:  %M1.Request,%M2.Response;
              %M2.Request,%M1.Response;
              %M1.Request,%M3.Response;
              %M3.Request,%M1.Response;
              %M1.Request,%P.Out;
              %M1.Response,%P.In

OFUN Get RETURNS T1
     PRE TRUE
     POST Get = %M1. Request.Window &
                %M1.Request.Flag = TRUE

OFUN Put(%Msg:T2)
     PRE TRUE
     POST %M1.Response.Window = Msg &
          %M1.Response.Flag = FALSE

VFUN Went RETURNS BOOLEAN
     PRE TRUE
     POST Went = %M1.Response.Flag

VFUN Came RETURNS BOOLEAN
     PRE TRUE
     POST Came' =  %M1.Request.Flag

OFUN Assign_UID (PUID: Process_UID_type)
     PRE TRUE
     POST %M1.UID = PUID

OFUN Assign_Index (Process_name:Machine_Index_Type)
     PRE TRUE
     POST %M1.Indx = Process_name


VFUN Send (Msg:T2)RETURNS BOOLEAN

     PRE %M1.Response.Flag=TRUE
            {to make sure previous message has been read.}

     POST Send=TRUE IFF %M1.Response.Window=Msg &
                        %M1.Response.Flag=TRUE

     BEHAVIOR

     %M1.Response.Window:=Msg
     %M1.Response.Flag:=FALSE
     WHEN
         %M1.Response.Flag --> Send::=TRUE

     END
```

```
VFUN Get_UID RETURNS Process_UID_Type
     PRE TRUE
     POST Get_UID'=%M1.UID
     BEHAVIOR

        {This function rturns the UID stored  in the
         given mailbox}

VFUN Get_Index RETURNS Machine_Index_Type
     PRE TRUE
     POST Get_Index' = %M1.Indx
     BEHAVIOR

        {This function RETURNS the process name stored
         the given mailbox.}

END  {Large_Mailbox}
```

3.2.16 Definition of abstract data type for Port_Msg

```
Port_Msg(S : TYPE) IS
Model  [ ID : UID_Type, Msg : s]
LET %Pm : Port_Msg (s)

VFUN Get_Msg RETURNS S
   PRE TRUE
   POST Get_Msg' := %PM.Msg

   BEHAVIOR
        {This function returns the msg part of
         the given Port_Msg.}

VFUN Get_UID RETURNS UID_Type
   PRE TRUE
   POST Get_UID' := %PM.ID

   BEHAVIOR
        {This function returns the ID part of
         the given Port_Msg.}


END Port_Msg
```

3.2.17 Definition of abstract data type for Outport

```
ACTIVE  Outport(T : TYPE) IS
MODEL [window : Port_Msg(T) , flag : BOOLEAN]
LET   %OP : Outport(T)  (%OP)
```

END Outport

3.2.18 Definition of abstract data type for Inport

```
ACTIVE Inport(T : TYPE) IS
MODEL [ window : Port_Msg(T), flag : BOOLEAN]
LET %IP : Inport(T)(%IP)

END Inport
```

3.2.19 Definition of abstract data type for Port

```
ACTIVE Port(T1 : Port_Msg, T2 : Port_Msg) IS
MODEL [In : Inport(T1) , Out : Outport(T2) ]
LET %P : Port
    %SM : Small_Mailbox
    %LM : Large_Mailbox

COMPLEMENTS %P.In, %SM.Response;
            %P.Out, %SM.Request;
            %P.In, %LM.Response;
            %P.Out, %LM.Request

OFUN Send(x : T2)
     PRE TRUE
     POST %P'.Out.window := x &
          %P'.Out.flag := FALSE

{We want flag = FALSE to indicate that information has been
 sent but not picked up yet by the matching INLET}
BEHAVIOR

   1- A call to send blocks until the flag indicate that a
      'get' has been done on its complements inlet.
   2- Then the send may proceed. Note that this send may
      proceed or another send may proceed.
   3- while the send is blocked it has no effect on the
      window or the flag.

END  Send


OFUN Receive(ID : UID_type) RETURNS T1
     PRE TRUE
     POST Receive'.ID = %P.In.window.Msg &
                        %P.In.flag = TRUE

{flag = FALSE indicates that information has been
 received from the inport.}

BEHAVIOR
```

```
    A call to receive blocks until a message with
    window.ID = ID arrives. Then receive proceeds
    to read that message.
END Receive

END Port

END  {TYPES DICTIONARY}
```

3.3 PROCEDURES DICTIONARY

3.3.1 Procedure Get_Modified_Objects

```
Get_Directly_Modified_Objects(Proc_UID : Process_UID_Type)
RETURNS PMDB_Response_Type

BEHAVIOR
     { This procedure sends a request to PMDB_Manager on PMDB_Iface
       to get the uid of all the objects which were modified by the
       given process or transaction .}

VARIABLES
     To_PMDB : PMDB_Request_Type

TEXT
To_PMDB.UID := Proc_UID
To_PMDB.Request := Modified_Object
WHEN
    PMDB_Iface.Send(To_PMDB) -->
    WHEN
        PMDB_Iface.Came -->
        Get_Modified_Object:= PMDB_Iface.Get
    END
END

END  Get_Modified_Object
```

3.3.2 Procedure Get_Children

```
Get_Children(Proc_UID : Process_UID_Type) RETURNS PMDB_Response_Type

BEHAIVIOR
      {This procedure sends a request to PMDB Manager to get
       the UIDs of all the children for given Proc_UID.}

VARIABLES
     To_PMDB : PMDB_Request_Type
```

```
TEXT
{Get UIDs of the children transactions and processes}
To_PMDB.Request := Parent_Children_Info
WHEN
    PMDB_Iface.Send(To_PMDB) -->
    WHEN
        PMDB_Iface.Came -->
        Get_Children := PMDB_Iface.Get
    END
END

END  Get_Children
```

3.3.3 Procedure Delete_From_PMDB

```
Delete_From_PMDB(Proc_UID : Process_UID_Type) RETURNS PMDB-Response_Type

BEHAIVOR
    {This procedure sends a request on PMDB_Iface to PMDB
     Manager to remove all the information about the given
     Proc_UID from the database.}

VARIABLE
     To_PMDB : PMDB_Request_Type

TEXT
To_PMDB.val.val := Proc_UID
WHEN
     PMDB_Iface.Send(To_PMDB) -->
     WHEN
         PMDB_Iface.Came -->
         Delete_From_PMDB := PMDB_Iface.Get
     END
END

END  Delete_From_PMDB
```

3.3.4 Procedure Add_To_PMDB

```
Add_To_PMDB(Rec : All_Rec_Type ARRAY) RETURNS PMDB_Response_Type

BEHAVIOR
     {This procedure sends a request to PMDB Manager
      to add the given record(s) to the database.
      Upon receiving of this request, PMDB Manager
      updates the database and Force it on the stable
      storage.}

VARIABLES
```

```
     To_PMDB : PMDB_Request_Type

TEXT
    To_PMDB.val.val.Op := Add
    To_PMDB.val.val.Param := Rec
    WHEN
        PMDB_Iface.Send(To_PMDB) -->
        WHEN
            PMDB_Iface.Came -->
            Add_To_PMDB := PMDB_Iface.Get
        END
    END

END  Add_To_PMDB
```

3.3.5 Procedure Get_LRP

```
Get_LRP(PUID : Process_UID_Type) RETURNS RP_Num

BEHAVIOR
     {This procedure sends a request to the PMDB Manager
      on PMDB_Iface to get the last recovery point for
      the given UID.}

VARIABLES

     To_PMDB : PMDB_Request_Type
TEXT
     To_PMDB.val.UID := PUID
     To_PMDB.val.Request := Get_LRP
     WHEN
         PMDB_Iface.Send (To_PMDB) -->
         WHEN
             PMDB_Iface.Came -->
             Get_LRP := PMDB_Iface.Get
         END
     END

END  Get_LRP
```

3.3.6 Discard_RP

```
Discard_RP(PUID : Process_UID_Type,
           RP1,RP2 : RP_Num)
RETURNS ACK

BEHAVIOR
     {This procedure sends a request on SS_Iface to
      Stable Storage Manager to discards all recovery
```

```
        points between and including RP1 & RP2 for the
        given procees.}
END  Discard_RP
```

3.3.7 Procedure Request_New_Process

```
Request_New_Process RETURNS Controller_To_CP

TEXT

    WHEN
        PM_Controller_Iface.Send(Action := New_Process) -->
        WHEN
            PM_Controller_Iface.Came -->
            Request_New_Process := PM_Controller_Iface.Get
        END
    END

END  Request_New_Process
```

3.3.8 Procedure Request_New_CP

```
Request_New_CP ( CP : Command_Machine_Type,
                 Index : INTEGER,
                 Work_Request : Appl_Req_Msg)
                 RETURNS Controller_To_CP

VARIABLES

     To_Controller : CP_To_Controller

TEXT
    To_Controller.Machine_Type := CP
    To_Controller.Descendent_Mbx_Indx := Index
    To_Controller.Caller_Type := My_Machine_Type
    To_Controller.Work_Request := Work_Request

    WHEN
        PM_Controller_Iface.Send(To_Controller)
        WHEN
            PM_Controller_Iface.Came -->
            Request_New_Process := Pm_Controller_Iface.Get
        END
    END

END  Request_New_CP
```

3.3.9 Procedure Create_Process_Record

```
Create_Process_Record(PUID : Process_UID_Type)
RETURNS Active_Process_Record

BEHAVIOR
        {This procedure creates an active process record
         for the process with the given UID.}
END
```

3.3.10 Procedure Create_Pc_Rec

```
Create_Pc_Rec(Parameter_list)
RETURNS Parent_Child_Info

BEHAVIOR
        {This procedure creates the parent child info
         record for the given process. The parameter
         list must contain all the required information.}
END
```

3.3.11 Procedure Create_PCB

```
Create_PCB(Params : Create_Param)
RETURNS PCB_Type

BEHAVIOR
        {This procedure loads the program & data (if any)
         into the main memory and create the process
         control block for the given process.}
END
```

3.3.12 Procedure Set_Timer

```
 Set_Timer(Time:INTEGER, TimerPort:Timer_Iface_Type)

TEXT

   WHEN
     TimerPort.Send([My_Process_ID, [ ], NULL, Time]) -->
   END

END {of Set_Timer}
```

3.3.13 Procedure Broadcast

```
 Broadcast(M:General_Msg_Type, Destination: Extended_UID_Array,
                       OutPort:OS_Iface_Type, My_ID:Process_UID_Type)

{This procedure broadcasts the message M to the  object  mangers of
the objects given by the parameter Destination.}

VARIABLES i:INTEGER

TEXT

  i:=Destination.lob

  DO (j:Destination.lob..Destination.hib) and (i=j) -->
     WHEN
       Outport.Send(<val::[My_ID, Obj_Manager(Destination(j),M>] --> SKIP
       {The function Obj_Manager returns the UID of the object manager
        of the object specified by the parameter UID}
     END
     i:=i+1
  OD

END {of Broadcast}
```

3.3.14 Procedure Create_RP_Data_Record

```
Create_RP_Data_Record(Proc_UID : Process_UID_Type,LRP: RP_Num)
RETURNS CHAR ARRAY

BEHAVIOR
        {This Procedure Creates the recovery point data
         Record (it contains the PCB and state of all the
         local variables of the process) and returns the
         name of the segment under which the record is
         stored.}}
END
```

3.3.15 Procedure Assign_Label

```
Assign_Label(Proc_UID : Process_UID_Type, LRP : RP_Num)
RETURNS Label_Type

BEHAVIOR
        {this procedure concatinates the given uid and
         lrp and returns a label under which the
```

```
        RP_Data_Record is to be stored.}
END
```

3.3.16 Procedure Get_Memory_Addr

```
Get_Memory_Addr ( RP_Data : CHAR ARRAY)
RETURNS Memory_Allocation

BEHAVIOR
        {This procedure returns the memory address &
         the length of the segment for the given
        name.}
END
```

3.3.17 Procedure Remove_Proc_Machine

```
Remove_Proc_Machine(UID : Process_UID_Type)
```

3.3.18 Procedure Get_All_Modified_Objects

```
BEHAVIOR
     {This procedure send a request on PM_Controller_Iface,
      to remove the process machine identified by Proc_UID
      from the Process POOL.}
END  Remove_Proc_Machine
Get_All_Modified_Objects(UID : Transaction_UID_Type)
RETURNS PMDB_Response_Type

BEHAVIOR
     {This procedure send a request on PMDB_Iface to
      get the  UID of all objects which were modified
      either directly or indirectly by given Transaction.}

END  Get_All_Modified_Objects
```

3.3.19 Procedure Get_PM_UID

```
Get_PM_UID (Host_ID : Host_ID_Type)
RETURNS Process_Manager_UID_Type

BEHAVIOR
     {This procedure returns the UID of Process_Manager
      of given host.}

END  Get_PM_UID
```

3.3.20 Procedure Terminate_Command_Processor

```
Terminate_Command_Processor

BEHAVIOR
     {This procedure sends a request on PM_Controller_Iface
      to terminate the command processor on which it is
      running.}

END  Terminate_Command_Processor
```

3.3.21 Procedure Check_Children_Status

```
Check_Children_Status(Children_UID : Process_UID ARRAY
RETURNS ACK

VARIABLES
        Children_Status : Transaction_Status_Type ARRAY
        i : INTEGER

TEXT
    Children_Status := Find_Children_Status(Children_UID)
    IF All_Complete(Children_UID) --> SKIP

     [] OTHERWISE -->
      Set_Timer(Timeout_Period, Timer_Iface)
      WHEN
          Interrupt.Came -->
          Check_Children_Status(Children_UID)
      END
    FI

END  Check_Children_Status
```

3.3.22 Procedure All_Complete

```
All_Complete(Children_Stat : Transaction_status_Type ARRAY)
RETURNS BOOLEAN

VARIABLES
       i : INTEGER
       Flag : BOOLEAN

TEXT
    Flag := TRUE
    i := 1
    DO (i..Children_Stat(dom))
      Children_Stat(i) <> completed AND Flag -->
```

```
      Flag := FALSE
      All_Completed := FALSE
    OD
END  All_Completed
```

3.3.23 Procedure Write_To_SS

```
Write_To_SS( L : Label_Type, Mem_info : Memory_Allocation)
RETURNS SS_PM_Msg

VARIABLES
      To_SS : PM_SS_Msg

TEXT
   To_SS.OP := Write
   To_SS.Param.Label := L
   To_SS.Param.Starting_Addr := Mem_Info.Starting_Addr
   To_SS.Param.Len := Mem_Info.Length

   WHEN
       SS_Iface.Send(To_SS) -->
       WHEN
           SS_Iface.Came -->
           Write_To_SS := SS_Iface.Get
       END
   END
```

3.3.24 Procedure Receive_Acks

```
END  Write_To_SS
Receive_Acks(UIDs : Process_UID_Type ARRAY,
             Timeout : INTEGER) RETURNS BOOLEAN

BEHAVIOR
     {This procedure receives either READY or ABORT messages from the object
      managers of the objects of given UIDs. This procedure either times out
      or returns when all object managers have responded.
      IF any of the messages is ABORT or Timeout then it returns TRUE,
      otherwise it returns   FALSE.

END  Receive_Acks
```

3.3.25 Procedure Get_Parent_Child_Info

```
Get_Parent_Child_Record(UID : Process_UID_Type)
RETURNS Parent_Child_Info
```

```
BEHAVIOR
     {This procedure searchs the Parent_Child_Info List for
      the given UID  and returns the record for that process.
     }
END  Get_Parent_Child_Record
```

3.3.26 Get_All_Descendent

```
Get_All_Descendents(UID : Process_UID_Type)
RETURNS Process_UID_Type ARRAY

BEHAVIOR
     {This procedure returns the UID of the
      descendents for given process/transaction.
     }
END  Get_All_Descendents
```

3.3.27 Procedure Update_Transaction_Status

```
Update_Transaction_Status(UID : Transaction_UID_Type,
                          Status : Transaction_Status_Type,
                          Force : BOOLEAN) RETURNS ACK

BEHAVIOR
     {This procedure sends a update request on PMDB_Iface to
      update the status of the given transaction and the updated
      database is forced on the stable storage if Force is TRUE.
     }
END  Update_Transaction_Status
```

3.3.28 Procedure Clear_Database

```
Clear_Database(UID : Process_UID_Type) RETURNS ACK
```

3.3.29 Procedure Signal

```
BEHAVIOR
     {This procedure sends a clear request on PMDB_Iface to
      remove all the information about the given UID from
      the database.
     }
END  Clear_Database
Signal(Parent :Process_UID_Type,Msg : General_Msg_Type)

BEHAVIOR
     This procedure sends a status message on the PMDB_Iface
     if the parent process is local then the status change is
     recorded locally in the parent's Parent_Child_Record
     otherwise a status update message is sent by the database
```

manager to the remote database manager where the parent is residing.

END Signal

3.4 REALIZATION DICTIONARY

3.4.1 Router_Machine

```
PUBLIC
        Router_TO_PM_Control_CMD: Small_Mailbox(PM_Router_CTRL_
                                  Msg,Router_PM_CTRL_Msg);
        Router_To_PM: Small_Mailbox(PM_Router_Msg,
                      Router_PM_Msg);
        Router_TO_Process: LIST(Large_Mailbox(Proc_Router_Msg
                           ,Router_Proc_Msg)
        Router_TO_OS : Small_Mailbox(OS_PM_Msg , PM_OS_Msg)

        Router_To_PMDB : Small_Mailbox(OS_PM_Msg, PM_OS_Msg)

OBJECTS

        { --- }

END {Router_Machine}
```

3.4.2 Process

```
PUBLIC

        Process_To_PM: Small_Mailbox(Response_Msg,
                       Appl_Req_Msg)
        Process_TO_PM_PSCMD: Small_Mailbox(Response_Msg,
                       Processor_Scheduler_Req_Msg)
        Process_TO_Router: Small_Mailbox(Router_Proc_Msg,
                                         Proc_Router_Msg)

OBJECTS

        PCB : PCB_Type

CONTROLLER

        {Executes the program code as per the Process Control Block
         and also executes the processor scheduler commands}

END {Process}
```

3.4.3 Timer

```
        PUBLIC

           Timer_Command_Iface: Small_Mailbox(Timer_Command, Timer_Response)
            Interrupt : Event_Rec OUTLET

        OBJECTS

                   { --- }

      END {Timer}
```

3.4.4 Machine Delete_Processor

```
Delete_Processor(My_Process_ID:Process_UID_Type,
                Remote:BOOLEAN,
                Requester:Process_UID_Type,
                Work_Request:Appl_Req_Msg)

PUBLIC
      PMDB_Iface:Port(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      SS_Iface:Port(Port_Msg(SS_PM_Msg),Port_Msg(PM_SS_Msg))
      MM_Iface:Port(Port_Msg(MM_PM_Msg),(Port_Msg(PM_MM_Msg))
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)

OBJECTS

      Timer_Interrupt : Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      Local_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                         Timer_Interrupt TO Interrupt)

      Response : Response_Msg
```

3.4.4.1 Procedure Delete_Remote_Process

```
 Delete_Remote_Process (Work_Request : Appl_Req_Msg,
                        My_Process_ID : Process_UID_Type)
 RETURNS Response_Msg

BEHAVIOR
     {This Procedure invokes a remote operation by sending
```

a request to Operation Switch, to delete a remote process. It waits until either it times out or gets a response back from remote host regarding to the result of the operation.

```
VARIABLES
      To_Router:General_Msg_Type
      Resp : Response_Msg
      Timeout_Period : INTEGER
      Receiver : Process_Manager_UID_Type

TEXT

   {Set the parameters for Invoke operation & send a message to
    Operation Switch thru Router to invoke this operation on the
    requested host.}
    TO_Router.val := Work_Request

    Receiver ::=Get_PM_UID(Work_Request.val.Param.Host_ID)
    Resp::=Invoke(My_Process_ID,Receiver, To_Router, OS_Iface)
    Set_Timer(Timeout_Period, Timer_Iface)
    IF Resp.tag = Success -->
       WHEN
           Timer_Interrupt.Came -->
           Delete_Remote_Process.val::=Time-Out

        [] OS_Iface.came -->
           Delete_Remote_Process:=(OS_Iface.Get).Operation.val
       END

     [] Resp.tag = Failure -->
       Delete_Remote_Process.val := Undefined_Error
    FI

  END {Delete_Remote_Process}
```

3.4.4.2 Procedure Delete_Local_Process

Delete_Local_Process(Proc_UId:Process_UID_Type) RETURNS Response_Msg

Algorithmic Description:

```
1. Suspend(PUID).
2. Get_list_Of_Modified_Objects(PUID)---> Set_Of_Object_UIDS
3. For Set_Of_Object_UIDS Do
3.1    Delete_Object_Versions(UID)---> {Successful,Unsuccessful}
4. Get_List_Of_Children(PUID)---> Set_Of_Children_UIDS
5. For Set_Of_Children_UIDS do
5.1    Delete_Process(UID)
6. Discard_Recovery_Points_On_SS(PUID)
```

```
BEHAVIOR
     {This procedure deletes the specified process or transaction
      and all its descendents regardless of thier status. It also
      broadcasts a message to all modified objects to discard all
      the changes which were made by this process.}

VARIABLES

      To_Controller : CP_TO_Controller
      k,i : INTEGER
      Resp : ACK
      Response : Response_Msg
      Modified_Obj : PMDB_Response_Type
      Children : PMDB_Response_Type


TEXT

{Destroys the Process}
WHEN
    PM_Controller_Iface.Send(Process_UID_Type) -->
   {The PM Controller removes all connections to the above process}
    WHEN
        PM_Controller_Iface.Came -->
        Resp:=PM_Controller_Iface.Get
    END
END

{Get the UIDs of all modified objects}

Modified_Obj := Get_All_Modified_Object(Proc_UID)

IF
  Modified_Obj.val.Modified_Objects(dom) > 0 -->
  Broadcast(<Other_Msg,['Delete_Version',0]>,Modified_Obj.Objects,
             OS_Iface,My_Process_ID)
 {This procedure broadcasts to all modified objects request to
  delete the object versions created by the process Pr_name}

 [] OTHERWISE --> SKIP
FI

Children_List := Get_Children(Proc_UID)

IF
  Children_List.val.Children(dom) > 0 -->
  No_of_Children := Children_List.val.Children(dom)
  i:=1
  DO (i <= No_of_Children) -->
    PUID := Children_List.Children(i)
    k := Extend_Mbx(Descendent_Iface)
    To_Controller.Machine_Type := Delete
```

```
    To_Controller.Descendent_Mbx_Indx := k
    To_Controller.Work_Request.OP := Delete_Process
    To_Controller.Work_Request.Param := Proc_UID
    WHEN
        PM_Controller_Iface.Send(To_Controller)-->
        WHEN
            PM_Controller_Iface.Came -->
            Response := PM_Controller_Iface.Get
        END
    END
    i := i + 1
  OD


{Update PM database.}

PMDB_Response := Delete_From_DB(Proc_UID)


{Discard all recover points on stable storage for
 deleted process}

Last_RP::Get_LRP(Proc_UID)
Discard_RPs(Proc_UID,0,Last_RP)


END    {Delete_Local_Process}

CONTROLLER

  IF Work_Request.tag≠Delete_Process -->
     Response.val::Illegal_Command
   [] OTHERWISE -->
     IF Work_Request.val.Param.Host_ID≠My_Host_ID -->
        Response ::Delete_Remote_Process(Work_Request, My_Process_ID)

      []  OTHERWISE -->
        Response ::Delete_Local_Process(Work_Request.val.Param)
     FI

  FI

  IF Remote -->
     OS_Iface.Send(Response)
   [] OTHERWISE -->
     Parent_Iface.Send(Response)
  FI
  Terminate_Command_Processor {this procedure destroys the cp.}

END   Delete_Processor Machine
```

3.4.5 Machine Create_Processor

```
Create_Processor(My_Process_ID:Process_UID_Type,
                 Remote:BOOLEAN,
                 Requester:Process_UID_Type,
                 Work_Request:Appl_Req_Msg)

PUBLIC
      PMDB_Iface:Port(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Small_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      SS_Iface:Port(Port_Msg(SS_PM_Msg),Port_Msg(PM_SS_Msg))
      MM_Iface:Port(Port_Msg(MM_PM_Msg),(Port_Msg(PM_MM_Msg))
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)

OBJECTS

      Timer_Interrupt : Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      Local_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                          Timer_Interrupt TO Interrupt)
```

3.4.5.1 Procedure Create_Remote_Process

```
 Create_Remote_Process (Work_Request : Appl_Msg_Type,
                        My_Process_ID : Process_UID_Type)
                        RETURNS Response_Msg
BEHAVIOR
     {This proedure invokes a remote operation by sending
      a request to the Operation Switch to create a process
      on a remote host. Then it waits until it either times
      out or gets the result of operation back from the
      remote host.}

VARIABLES
        To_Router:General_Msg_Type
        Response:Response_Msg
        Resp : ACK
        Receiver : Process_Manager_UID_Type

TEXT

   {Set the parameters for Invoke operation & send a message to
    Operation Switch thru Router to invoke this operation on the
    requested host.}
    TO_Router.val := Work_Request
```

```
Receiver := Get_PM_UID(Work_Request.val.Param.Host_ID)
Resp::=Invoke(My_Process_ID, Receiver,To_Router, OS_Iface)
Set_Timer(Timeout_Period, Timer_Iface)
IF Resp.tag = Success -->
   WHEN
       Timer_Interrupt.came -->
       Create_Remote_Process.val::=Time-Out

    []  OS_Iface.came -->
       Create_Remote_Process::=(OS_Iface.Get).Operation.val
                                    {Create_Remote_Process_Msg type}
   END

 [] Resp.tag = Failure -->
   Create_Remote_Process.val := Undefined_Error
FI

END {Create_Remote_Process}
```

3.4.5.2 Procedure Create_Local_Process

```
Create_Local_Process(Work_Request : Appl_Req_Msg)
RETURNS Response_Msg

Algorithmic Description:

1.  Assign_UID ---> TUID
2.  Create_PCR (TUID) ---> PCR_record
3.  Insert_PCR (PCR_record) ---> {successful, unsuccessful}
4.  Create_PCB (TUID, Prog, [Data]) ---> PCB_record
5.  Insert_to_Parent_child_table (TUID) ---> {successful, unsuccessful}
6.  Set_Map_field (TUID, 0) ---> {successful, unsuccessful}
7.  Establish_Recovery_Point (TUID) ---> RP_Num
8.  Run_PCB (PCB_record) ---> {successful, unsuccessful}
9.  Return (status).
 BEHAVIOR
     {This procedure creates either a new process in response to
      'Create_Process' or a new transaction in response to
      'Create_Transaction' or 'Begin_Transaction'.
      It also creates all the records which are needed to be kept
      in PM_Database for the newly created process or transaction.
      And after it establishes the first recovery point, it sends
      a request to PM controller to start running the process.}


 VARIABLES
         Pr_Rec:Active_Process_Record
         PC_Rec: Parent_Child_Info
         Proc_UID : Process_UID_Type
         PCB : PCB_Type
```

```
        Controller_Resp : Controller_To_CP
        PMDB_Resp : PMDB_Response_Type
        Temp_Rec : All_Rec_Type ARRAY
        Response : Response_Msg
        To_Controller : CP_To_Controller
        To_Parent : Response_Msg

TEXT

{Send request to the controller to create a new process, connect it to the
 appropriate mailboxes, and then return its UID}
  Controller_Resp := Request_New_Process
  IF Controler_Resp.tag = t2 -->
     Proc_UID :=Controller_Resp.val

     IF Work_Request.val.OP = Create_Process -->
       {Create active Process record}
        Pr_Rec::=Create_Process_Record (Proc_UID)

      [] OTHERWISE -->
       {Create active transaction record}
        Pr_Rec := Create_Transactin_Record(Proc_UID)
     FI

    {Create parent_child_info record
     Pc_Rec::=Create_Pc_Rec(Params)

    IF Work_Request.val.OP = Create_Process  OR
       Work_Request.val.OP = Create_Transaction -->
       {Create process control block}
       PCB := Create_PCB(Params)

     [] OTHERWISE --> SKIP
    FI

    {Establish the 1st recovery point for the new process}
     Work_Request.OP::=Establish_Recovery_Point
     Work_Request.Param := Proc_UID
     J::=Descendent_Iface.Hi_Extend
     Controller_Resp::=Request_New_CP(ERP,J,Work_Request)

     WHEN
       Descendent_Iface.Element(J).came -->
       Response::=Descendent_Iface.Element(J).Get
     END


     IF  Response.tag = Success -->
        {Send request message to the Database_Iface; wait for the response}
```

```
        Temp_Rec(1).val := Pr_Rec
        Temp_Rec(2).val := Pc_Rec
        PMDB_Resp := Add_To_PMDB(Temp_Rec)

        IF PMDB_Resp.val = Success -->
           {start running the process}
           To_Controller.Action := Run_Process
           To_Controller.Param := PCB
           WHEN
               PM_Controller_Iface.Send(To_Controller)-->
               WHEN
                   PM_Controller_Iface.Came -->
                   Controller_Resp := PM_Controller_Iface.Get
               END
           END
           IF Controller_Resp.val = Failure -->
              Remove_Proc_Machine(Proc_UID)
              Create_Local_Process.val := Undifined_Error

            [] OTHERWISE -->
              To_Parent.val := Work_Request
              WHEN
                  Parent_Iface.Send(To_Parent) --> SKIP
              END
           FI

     [] OTHERWISE -->
        Create_Local_Process.val := Undefined_Error
        Discard_RP(proc_UID,0,0)
        Remove_Rroc_Machine(Proc_UID)
  FI

 [] OTHERWISE -->
  Create_Local_Process.val := Undefined_Error
FI
END {Create_Local_Process}


CONTROLLER

  IF Work_Request.tag≠Create_Process -->
     Response.val:=Illegal_Command
   [] OTHERWISE -->
     IF Work_Request.val.Param.Host_Hint≠My_Host_ID -->
        Response:=Create_Remote_Process(Work_Request, My_Process_ID)

      []  OTHERWISE -->
        Response:=Create_Local_Process(Work_Request)
     FI

  FI
```

```
  IF Remote -->
     OS_Iface.Send(Response)

   [] OTHERWISE -->
     Parent_Iface.Send(Response)
  FI

 Terminate_Command_Processor

END  Create_Processor Machine
```

3.4.6 Machine PM_Database_Manager

```
PUBLIC
          SS_Iface:Port(Port_Msg(SS_PM_Msg),(Port_Msg(PM_SS_Msg))
          Database_Port:Small_mailbox(Port_Msg(PMDB_Request_Type),
                        Port_Msg(PMDB_Response_Type))
          OS_Iface : Small_Mailbox(Invoke_Msg_Type, Invoke_msg_Type)

OBJECTS

            {PM Database:}

            Active_Process_List: LIST (Active_Process_Record)
            Parent_Child_Info_List: LIST (Parent_Child_Info)
            Directly_Modified_Object_List: LIST (Directly_Modified_Objects)
            Current_Operation_List: LIST (Current_Operation_Info)
            PMDB_Log_Buffer:LIST (PMDB_Modify_Operations)

CONTROLLER

   {Receives request messages from the Database_Port. The request messages
    are either querries or updates. The request messages also bear the
    Process UID of the caller.  The controller sends the responses to the
    Database_Port,  which are then received by the process that originated
    the request. This machine also interfaces with both the secondary memory
    port, in order to periodically save the database on the stable storage,
    and the Operation Switch, in order to inform the remote hosts of the
    changes that affect thier databases.}

END PM_Database_Manager Machine
```

3.4.7 Machine Port_Multiplexer

```
Port_Multiplexer( T1 : TYPE , T2 : TYPE)
```

```
PUBLIC
      Device_Iface : Small_Mailbox(Port_Msg(T1),Port_Msg(T2))
      Iface : Small_Mailbox(Port_Msg(T2),Port_Msg(T1))

OBJECTS
       UID : Process_UID_Type
       Response : Port_Msg(T1)


Procedure  Attach_UID(UID : Process_UID_Type,
                      Msg : T1) RETURNS Port_Msg

BEHAVIOR
     {This procedure attachs the UID of the requester
      to the recieved msg.}

END  Attach_UID

CONTROLLER

      WHENEVER
         Iface.Came -->
         UID := Iface.Get.Get_UID
         Device_Iface.Send(Iface.Get.Get_Msg)

       [] Device_Iface.Came -->
         Response := Attach_UID(UID, Device_Iface.Get)
         Iface.Send(Response)
      END

END  Port_Multiplexer
```

3.4.8 Machine End_Trans_Processor

```
End_Trans_Processor(My_Process_ID:Process_UID_Type,
                    ERP:BOOLEAN,
                    Requestor:Process_UID_Type,
                    Work_Request:Appl_Req_Msg)

PUBLIC

      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)

OBJECTS

      Timer_Interrupt IS Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
```

```
                                  Timer_Command)
      Local_Timer:Timer::=(Timer_Iface TO Timer_Command_Iface,
                           Timer_Interrupt TO Interrupt)
```

3.4.8.1 Procedure Commit

Commit(TUID :Transaction_UID_Type)

BEHAVIOR

The execution of the End_Transaction command is the commit point for the transaction. For an outermost transaction, execution of the End_Transaction command means permanence of all updates made within this transaction and releasing of the locks on the updated objects. For a nested transaction, execution of the End_Transaction command means only a conditional commitment that is dependent upon the commitment of its enclosing transaction.

Description of the End_Transaction Protocol:

In this design we will make the following assumptions:

1) All update operations on remote or local objects follow the two-phase commit protocol.

2) All nested transactions follow the one-phase commit protocol.

3) Every transaction maintains a list of all object that have been directly or indirectly modified by it. An object is said to be indirectly modified by a transaction when it is modified only within one of its nested transactions. Any object that is updated within a transaction by invoking a local or remote procedure call is said to be directly modified by the transaction.

4) We use the presumed-abort protocol if no information is present about the transaction status in the Process Manager's database.

End_Transaction protocol:

{abort}.

3) If all responses are ACKs then execute the following protocol:

3.1) Record this operation in the Current_Operation_List;

3.2) If ERP option then establish recovery point for the parent process;

4) if the transaction is
   top-level and sequential -->
      (a) Force write the COMMIT status record in the database;

```
   (b) Send COMMIT messages to all Object Managers of modified objects;
   (c) Resume the parent process;
   (d) Receive ACKs for the commit messages from the Object Managers;
   (e) When all ACKs are received, delete all information about
       the transaction from the database; }

top-level and concurrent -->
   (a) Force write COMPLETED status for the transaction;
   (b) Send COMPLETED messages to all Object Managers of modified objects;
   (c) Send DONE signal to the parent transaction;

   { Background activity executed by the Commit/Abort command processor:
        Wait for the COMMIT/ABORT from the parent process;
        If
        COMMIT command -->
           Force write COMMIT record in the database;
           Send COMMIT messages to all Object Managers of the
           modified objects;
        ABORT command -->
           Send ABORT messages to all Object Managers;
           Delete all information for the transaction from
           the database;
        fi;

        Wait for the ACKs for the COMMIT messages;
        if
        all ACKs received -->
            Delete all information for the transaction from
            the database;
        fi; }

Nested and Sequential -->

   (a) Send COMPLETED message to all modified objects;
   (b) Force write COMPLETED record in the database;
   (c) Append the list of the modified objects to the parent transaction;
   (d) Append the list of the descendent transactions to the parent's
   list;
   (e) Resume the parent process;

   { Background activity executed by the Commit/Abort command processor:
        Wait for the COMMIT/ABORT command from the parent;
        When such a command is received, delete all
        information about the transaction from the database;
        Send an ACK for the COMMIT command; }

nested and concurrent -->

   (a) Force write COMPLETED record in the database;
   (b) Send COMPLETED message to all Object Managers of the modified
   objects;
   (c) Append the list of the modified objects to the parent transaction;
```

```
   (d) Append the list of the descendent transactions to the parent's
   list;
   (e) Send DONE signal to the parent transaction;

   { Background activity executed by the Commit/Abort command processor:
       Wait for the COMMIT/ABORT command from the parent;
       Delete all information for the transaction from the database;
       Send ACK for the COMMIT command; }
fi

5) EXIT
6) Invoke the ABORT command;
```

VARIABLES

```
          Current_Op_Rec: Current_Operation_Info
          Modified_Obj_UIDS : PMDB_Response_Msg
          msg:Invoke_Msg_Type
          Parent:Process_UID_Type
          PC_Rec:Parent_Child_Info
          Response : Response_Msg
          Abort_Signal : BOOLEAN
          All_Modified_Obj : PMDB_Response_Msg
          Descendant_Transaction : PMDB_Response_Msg
```

TEXT

```
{Delete the transaction process. The information about this transaction
 in the PM database, and its recovery points are still intact}

 PM_Controller_Iface.Send(<Service_Call, [Action::=Delete_Process,
                              Param:= <tag2, TUID>]
                            >)

{add (TUID,END_TRANSACTION) to current_operation_table}

Current_Op_Rec.PUID ::=TUID
Current_Op_Rec.op::=End_Transaction

PMDB_Iface.Send(<Update, <Add_Modify,
                           [OP:=Add, Param:=(COI,Current_Op_Rec)]
                          >
                 >)

{If ERP option then establish  a recovery point for the parent process}

If ERP_Option then {establish recovery point for the parent process by
                    creating an ERP_Processor via the PM Controller};
```

```
{Get list of objects directly  modified by TUID}
Modified_Obj_UIDS::=Get_Directly_Modified_Objects(TUID)

{Get list of all objects (directly or indirectly) by TUID }
All_Modified_Obj:=Get_All_Modified_Obj(TUID)

Descendent_Transactions::=Get_All_Descendent(TUID)

{Send PREPARE message to the object managers ofdirectly modified objects}
Broadcast('PREPARE', Modified_Obj_UIDS.Modified_Object,
           OS_Iface, My_Process_ID)

{wait to receive READY/ABORT messages}
Abort_Signal::=Receive_Acks(Modified_Obj_UIDS.Modified_Object, Timeout_Period)


IF Abort_Signal -->
   K::=Extend(Descendent_Iface)
   WHEN
     PM_Controller_Iface.Send(<Create,
                                [Machine_Type::=Abort,
                                 Descendent_Iface_Indx::=K,
                                 Caller_Type::=End_Trans,
                                 Work_Request::=<UID_param,
                                                 [OP::=Abort, Param::=TUID]>
                                                 ]>)

      -->

      WHEN
          PM_Controller-Iface.Came -->
          WHEN
              Descendent_Iface.Element(K).Receive(Response) -->
              WHEN
                  Parent_Iface.Send(Response)
              END
          END
      END
  END

  ] OTHERWISE {Abort_Signal is set to false} -->


{Search parent_child_info_LIST to get Map-field}
PC_Rec::=Get_Parent_Child_Record(TUID)

Parent::=PC_Rec.Parent_UID

IF (PC_Rec.Top_Level) AND (PC_Rec.Map_Field=Sequential) -->
   {Top-level and Sequential Transaction}
   Force := TRUE
   Update_Transaction_Status(TUID, Committed, Force)
```

```
{Resume the parent process by sending the command message to the controlle
PM_Controller_Iface.Send(<Service_Call, [Action::=Run_Process,
                           Param::= <tag2, Parent>]
                          >)

 Broadcast('COMMITTED',All_Modified_Obj.Modified_Object,
            OS_Iface,My_Process_ID)
 Broadcast('COMMITTED',Descendent_Transactions.Children,
            OS_Iface,My_Process_ID)
 Set_Timer(timeout_period, Timer_Iface)

{Wait to receive ACKs from all modified objects when all ACKs have
 been received delete all information about this transaction
 from the database.}

 DO (All_Modified_Obj.Modified_Object.dom <>0) OR
    (Descendent_Transactions.Children.dom<>0) -->
   WHEN OS_Iface.Came --> msg::=OS_Iface.Get
             IF msg.Operation.val.response='ACK' -->
                IF It_Is_Transaction(msg.Operation.val.Obj) -->
                   Delete(msg.Operation.val.Obj, Descendent_Transaction
                [] OTERHWISE -->
                   Delete(msg.Operation.val.Obj,
                          All_Modified_Obj.Modified_Object)
             FI
       [] Timer_Interrupt.Came -->
             Broadcast('COMMITTED',All_Modified_Obj.Modified_Object,
                       OS_Iface,My_Process_ID)
             Broadcast('COMMITTED',Descendent_Transaction,
                        OS_Iface,My_Process_ID)
             Set_Timer(timeout_period, Timer_Iface)
   END
OD

Clear_Database(TUID)


[] (PC_Rec.Top_Level) and (PC_Rec.Map_Field=Concurrent) -->
   {Top-level and Concurrent}

    Force := TRUE
    Update_Transaction_Status(TUID, Completed, Force)

    Signal(Parent, DONE)

   {A Commit_Processor will be created in response to a COMMIT command
    from the parent process}


[] (not PC_Rec.Top_Level) and (PC_Rec.Map_Field=Sequential) -->
   {Nested and Sequential transaction}
```

```
      Force := TRUE
      Update_Transaction_Status(TUID, Completed, Force)

     {Send COMPLETED message to all modified objects}
      Broadcast('COMPLETED',All_Modified_Obj.Modified_Object,
                 OS_Iface, My_Process_ID)

    {Append the list of modified objects to  the parent transaction, and
     append the list of the descendent transactions to the parent's database;
     Append_Modified_Obj_List(All_Modified_Obj.Modified_Object, Parent)
     Append_Descendent_List(Descendent_Transactions.Children, Parent)

    {Resume the parent process by sending the command message to the controlle
     PM_Controller_Iface.Send(<Service_Call, [Action::=Run_Process,
                                 Param:= <tag2, Parent>]
                                >)

[] (not PC_Rec.Top_Level) and (PC_Rec.Map_Field=Concurrent) -->
    {it is a  nested concurrent transaction}

     Force := TRUE
     Update_Transaction_Status(TUID, Completed, Force)

    {Send COMPLETED message to all object managers}
     Broadcast('COMPLETED', All_Modified_Obj.Modified_Object,
                 OS_Iface, My_Process_ID)

    {Append the list of modified objects to  the parent transaction, and
     append the list of the descendent transactions to the parent's list}

     Append_Modified_Obj_List(All_Modified_Obj.Modified_Object, Parent)
     Append_Descendent_List(Descendent_Transactions.Children, Parent)


 FI
 Clear_Current_Op_Table(TUID)

END {of Commit procedure}

CONTROLLER

  TUID ::Work_Request.val.Param
  Commit(TUID)
  Terminate_Command_Processor

END {of END_TRANSACTION}
```

3.4.9 Machine Abort_Processor

```
Abort_Processor(My_Process_ID:Process_UID_Type,
                Remote:BOOLEAN,
                Requester:Process_UID_Type,
                Work_Request : Appl_Req_Msg
```

PUBLIC

```
     PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
     Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
     PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
     OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
     Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)
```

OBJECTS

```
     Timer_Interrupt IS Event_Rec INLET
     Timer_Iface:Small_Mailbox(Timer_Response,
                               Timer_Command)
     Local_Timer:Timer::=(Timer_Iface TO Timer_Command_Iface,
                          Timer_Interrupt TO Interrupt)
     Request:Process_UID_Type
```

3.4.9.1 Procedure Abort

Abort

BEHAVIOR

The Abort operation terminates the execution of the current block and restores the state of the local variables and global objects to their values before the beginning of Transaction and continues execution with the statement immediately following the End_Transaction statement of the aborted transaction. If Abort command is used within a process, it terminates the process.

This procedure executes the following steps to carry out the Abort operation:

I) Transaction wants to abort itself
   1_ Delete the process.
   2_ Restore the parent.
   3_ Change the status of the transaction or process to be aborted.
   4_ Send an Abort message to all the modified objects and its descendents.

II) Transaction wants to abort another transaction
   1_ Delete the process.

```
VARIABLES

          Current_Op_Rec: current_operation_Info
          Modified_Obj_UIDS : PMDB_Response_Msg
          msg:Invoke_Msg_Type
          Parent:Process_UID_Type
          PC_Rec:Parent_Child_Record
          Transaction_Descendents : PMDB_Response_Msg

TEXT

Search parent_child_info_LIST to get Map-field}
PC_Rec::=Get_Parent_Child_Record(TUID)
Modified_Obj_List:=Get_All_Modified_Obj(TUID)
Descendent_Transactions := Get_All_Descendent(TUID)
Parent::=PC_Rec.Parent_UID

IF Requester=TUID -->
   {A transaction wants to abort itself}

   PM_Controller_Iface.Send(<Service_Call, [Action::=Delete_Process,
                                            Param::=<tag2,TUID>]
                            >)
   Restore(Parent) {This procedure will restore the status of  all
                    local state varriables of the parent process}

   {Restart the parent process}
   PM_Controller_Iface.Send(<Service_Call, [Action::=Run_Process,
                                            Param::=<tag2,Parent>]
                            >)

[] OTHERWISE {requestor wants to abort some transaction} -->

   IF Status(TUID)<>COMPLETED -->
      PM_Controller_Iface.Send(<Service_Call, [Action::=Delete_Process,
                        Param::=<tag2,TUID>]
                            >)
   FI
FI

Update_Transaction_Status(TUID, 'ABORTED')
Broadcast('ABORTED',All_Modified_Obj.Modified_Object,
          OS_Iface,My_Process_ID)
Broadcast('ABORTED',Descendent_Transaction,
          OS_Iface,My_Process_ID)
Clear_Database(TUID)

END  Abort
```

```
    Broadcast('ABORTED',Descendent_Transactions,
                        OS_Iface,My_Process_ID)
    Clear_Database(TUID)

END  Remote_Abort
```

3.4.10 Machine Commit_Processor

```
Commit_Processor(My_Process_ID:Process_UID_Type,
                 Remote:BOOLEAN,
                 Requester :Process_UID_Type,
                 Work_Request : Appl_Req_Msg

PUBLIC

      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)

OBJECTS

      Timer_Interrupt IS Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      Local_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                          Timer_Interrupt TO Interrupt)
      Request:Process_UID_Type
```

3.4.11 Procedure Commit_Protocol_Terminator

```
  Commit_Protocol_Terminator(TUID :Transaction_UID_Type)
```

BEHAVIOR

The Commit operation makes all the updates which have been performed by a transaction permanent.

The followings are the steps which are executed by Commit_Protocol_Terminator :

I) Top level concurrent transaction AND command=Commit:
  1_ Change the transaction status to Committed and Force the updated database on stable storage.
  2_ Broadcast Commit message to all the modified objects and transaction's descendents.
  3_ Wait to receive ACK back from all the descendents and modified objects.
  4_ Clear the database.

```
CONTROLLER
   TUID := Work_Request.val.Param
   IF TUID <> My_Host_ID --->
      Response:= Remote_Abort( Work_Request, My_Process_ID)

   [] OTHERWISE
      Abort
   FI

   Terminate_Command_Processor

END {of Abort_Processor}
```

3.4.9.2 Procedure Remote_Abort

Remote_Abort

Algorithmic Description:

Remote_Abort follows following steps:

1. Signal completion to the parent process/transaction.
2. Send ABORT messages to type managers of all modified objects.
3. Send ABORT messages to Process Managers of the nested transactions.
4. Clear from PMDB all info related to the transaction UID.

BEHAVIOR

The Remote_Abort terminates the execution of the transaction given by TUID. It sends abort messages to the type managers of all modified objects and to the Process Managers of the nested transactions. It signals completion to the parent process/transaction. It clears from PMDB all info related to TUID.

VARIABLES

```
Modified_Obj_List: PMDB_Response_Msg
Parent: Process_UID_Type
PC_Rec: Parent_Child_Record
Transaction_Descendents: PMDB_Response_Msg
```

TEXT

```
PC_Rec:= Get_Parent_Child_Record(TUID)
Modified_Obj_List:= Get_All_Modified_Obj(TUID)
Descendent_Transactions:= Get_All_Descendent(TUID)
Parent:= PC_Rec.Parent_UID
Update_Transaction_Status(TUID,'COMPLETED')
Broadcast('ABORTED',Modified_Obj_List,
                    OS_Iface,My_Process_ID)
```

II) Nested sequential or concurrent transaction AND command=Commit:
1_ Send ACK message to the parent.
2_ Clear the database.

VARIABLES

```
        Current_Op_Rec: current_operation_Info
        Modified_Obj_UIDS: UID_type ARRAY
        msg:Invoke_Msg_Type
        Parent:Process_UID_Type
        PC_Rec:Parent_Child_Record
```

TEXT

```
{add (TUID,Commit) to Current_Operation_Table}

Current_Op_Rec.PUID ::=TUID;
Current_Op_Rec.op::=Commit;

PMDB_Iface.Send(<Update, <Add_Modify,
                          [OP::=Add, Param::=(COI,Current_Op_Rec)]
                          >
                >)


Search parent_child_info_LIST to get Map-field}
PC_Rec::=Get_Parent_Child_Record(TUID)
Modified_Obj_List::=Get_All_Modified_Obj(TUID)
Descendent_Transaction(TUID)
Parent::=PC_Rec.Parent_UID

IF (PC_Rec.Top_Level) AND (PC_Rec.Map_Field=Concurrent) -->
   {Top-level and Concurrent Transaction}
 IF Command=COMMIT -->
      Force := TRUE
      Update_Transaction_Status(TUID, Committed, Force)

      OS_Iface.Send([Sender::=My_Process_ID,
                     Receiver::=Requester
                     Operation::=<Other_Msg, [Response::='ACK',
                                              Obj::=TUID]
                                 >
                    ])

      Broadcast('COMMITTED',All_Modified_Obj.Modified_Object
                ,OS_Iface,My_Process_ID)
      Broadcast('COMMITTED',Descendent_Transactions.Children,
                 OS_Iface,My_Process_ID)
      Set_Timer(timeout_period, Timer_Iface)
```

```
        {Wait to receive ACKs from all modified objects when all ACKs have
        been received delete all information about this transaction
        from the database.}

         DO (All_Modified_Obj.Modified_Object.dom <>0) OR
            (Descendent_Transactions.Children.dom<>0) -->
           WHEN
               OS_Iface.Came --> msg::=OS_Iface.Get
                 IF msg.Operation.val.response='ACK' -->
                    IF It_Is_Transaction(msg.Operation.val.Obj) -->
                       Delete(msg.Operation.val.Obj, Descendent_Transaction)
                     [] OTERHWISE -->
                       Delete(msg.Operation.val.Obj,
                              All_Modified_Obj.Modified_Object)
                    FI
               [] Timer_Interrupt.Came -->
                      Broadcast('COMMITTED',All_Modified_Obj.Modified_Object,
                                OS_Iface ,My_Process_ID)
                      Broadcast('COMMITTED',Descendent_Transaction
                                ,OS_Iface,My_Process_ID)
                      Set_Timer(timeout_period, Timer_Iface)
              END
           OD


           Clear_Database(TUID) {Clear all information about TUID from the
                                 database}

     FI


[] (not PC_Rec.Top_Level) -->
      {Nested and sequential/concurrent transaction}

      IF Command=COMMIT -->
         {Send ACK and clear the database}
         OS_Iface.Send([Sender::=My_Process_ID,
                  Receiver::=Requester
                  Operation::=<Other_Msg, [Response::='ACK',
                                          Obj::=TUID]
                                   >
                     ])
       FI
       Clear_Database(TUID) {Clear all information about TUID from the
                                  database}


 [] Clear_Database(TUID)
FI

END   Commit_Protocol_Terminator
```

```
CONTROLLER
    TUID := Work_Request.val.Param
    Commit_Protocol_Termainator(TUID)
    Terminate_Command_Processor;

END  Commit_Processor
```

3.4.12 Machine Rollback_Processor

```
Rollback_Processor(My_Process_ID:Process_UID_Type,
                   TUID : Process_UID_Type,
                   RP_Number:INTEGER)

PUBLIC

      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      Descendent_Iface:LIST(Large_Mailbox((Response_Msg, Appl_Req_Msg)

OBJECTS

      Timer_Interrupt IS Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      Local_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                          Timer_Interrupt TO Interrupt)
```

3.4.12.1 Procedure Rollback

Rollback

BEHAVIOR

The Rollback operation within a process restores the state of all the local objects to their values which they possessed at the time the RP_num was established. The changes on global objects, which have been made by transactions within that process, remain permanent if the transactions performing those are committed; otherwise, they are resotred to their values that they possessed at the time the RP_num was established.

The Rollback operation within a transaction restores the state of all the local and global objects to their values that they had at the time the RP_num was established.
The following outlines the steps which must be taken to perform the Rollback operation:

I) Transaction Type:
   1_ Add the operation to the current operation info.
   2_ Get the Time_stamp for the time the recovery point was established.

3_ Broadcast a message to all the modefied objects to delete all the versions which were created after the that time.
4_ Broadcast an Abort message to the all the children which were created after that time.
5_ Wait to rceieve ACK from all modified objects and children.
6_ Restore the recovery point data .
7_ Update the the last recovery point number.

II) Process Type:
1_ Add the operation to the current operation info.
2_ Restore the recovery point data.

```
VARIABLES
     Current_Op_Rec : Current_Operation_Info
     Ts : INTEGER
     Modified_Obj : PMDB_Response_Msg
     New_Children : PMDB_Response_Msg
     Timeout_Period : INTEGER
     PCB_Rec : PCB_Type
     LRP : RP_Number

TEXT
Current_Op_Rec.PUID ::=Proc_UID
Current_Op_Rec.op::=Rollback

PMDB_Iface.Send(<Update, <Add_Modify,
                           [OP:=Add, Param:=(1,Current_Op_Rec)]
                          >
                 >)

TS::=Time_stamp(RP_Number, Proc_UID) {This procedure returns the time-stamp
                                       of the recovery point number RP_Number
                                       for Proc_UID}

IF It_Is_Transaction(Proc_UID) -->
   Modified_Obj::=Get_Modified_Obj(Proc_UID) {All directly modified objects}
   New_Children::=Get_New_Children(RP_Number, Proc_UID)
                 {This procedure returns the children transactions created
                  by Proc_UID after establishing the recovery point}

   IF Modified_Obj.dom<>0 -->
      Broadcast( <Other_Msg, ['Delete_Version', TS]>,
                 Modified_Obj, OS_Iface, My_Process_ID)
   FI

   IF New_Children.dom<>0 -->
      Broadcast('ABORT', New_Children, OS_Iface, My_Process_ID)
   FI

    Set_Timer(timeout_period, Timer_Iface)
```

```
    {Wait to receive ACKs from all modified objects, and the children
     transactions that are to be aborted.}

     DO (Modified_Obj.dom <>0) OR (New_Children.dom<>0) -->
       WHEN OS_Iface.Came --> msg::=OS_Iface.Get
                   IF msg.Operation.val.response='ACK' -->
                      IF It_Is_Transaction(msg.Operation.val.Obj) -->
                           Delete(msg.Operation.val.Obj, New_Children)
                           {This procedure removes the UID of the object
                            that sent this ACK}
                      [] OTHERWISE -->
                           Delete(msg.Operation.val.Obj,Modified_Obj)
                           {This procedure removes the UID of the object
                            that sent this ACK}
                   FI
          [] Timer_Interrupt.Came -->
                   Broadcast( <Other_Msg, ['Delete_Version', TS]>,
                              Modified_Obj, OS_Iface, My_Process_ID)
                   Broadcast('ABORT',New_Children,OS_Iface,My_Process_ID)
                   Set_Timer(timeout_period, Timer_Iface)
       END
    OD

 [] OTHERWISE {non-transaction process} --> SKIP

 FI

 PCB_Rec:= Restore_Recovery_Point(Proc_UID, RP_Number)
          {This procedure loads the recovery point data in the primary
           memory, prepares the process control block and returns
           this as the result}

 LRP::=Last_Recovery_Point(Proc_UID)

 IF RP_Number<>LRP -->
    Discard_Recovery_Point(Proc_UID, RP_Number, LRP)
 FI

 PM_Controller_Iface.Send(<Service_Call, [Action=Run_Process,,
                                          Param::=PCB_Rec]
                          >)

 Clear_Current_Op_Table(Proc_UID)


 END {of Rollback }

 CONTROLLER

    Rollback
    Terminate_Command_Processor
```

```
END  { Abort Processor }
```

3.4.13 Machine ERP_Processor

```
ERP (My_Process_ID:Process_UID_Type,
     Requester : Process_UID_Type
     Work_Request:Appl_Req_Msg)

PUBLIC
      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      PM_Controller_Iface:Small_Mailbox(Controller_to_CP,CP_to_Controller)
      OS_Iface:Large_Mailbox(Invoke_Msg_Type, Invoke_Msg_Type)
      SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)
      MM_Iface:Port(MM_PM_Msg,PM_MM_Msg)

OBJECTS

      Timer_Interrupt IS Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      PM_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                       Timer_Interrupt TO Interrupt)

      Response : Response_Msg
```

3.4.13.1 Procedure_Establish_RP

```
Establish_RP(Proc_UID :Process_UID_Type)
RETURNS Response_Msg
```

Algorithmic Description:

```
1.0  UID <----- Get_Invoker_UID;
2.0  PCB <----- Get_PCB (UID);
3.0  LRP <----- Get_LRP (UID);
4.0   LRP <----- LRP  + 1:

     {The value for the LRP in the Process Manager Database will be
     changed after the operation is done successfully. The above LRP is
     just a temporary variable.}

5.0  Label <----- Assign_Label (UID, LRP);
6.0  (Starting_addr, Len) <----- Create_RP_Data_Rec (Data, PCB);
7.0  Write (Label, Starting_addr, len);
8.0  LRP <----- Update_LRP (UID) -Log;

     {'Log' indicates that the changes to the Process_Manager_Database are
     to be recorded on the PMDB_Log_Buffer (maintained in PM_database).}
9.0  Append (PMDB_Log_Buffer, Differential_File);
```

```
10.0 Resume the invoker
VARIABLES
          PCB:PCB_Type
          LRP:RP_Num
          L:Label_Type
          RP_Data:CHAR ARRAY
          MM_Info:Memory_Allocation
          Param:RW_Param
          Controller_Resp : Controller_To_CP
          SS_Resp : SS_PM_Msg
          Children_List : PMDB_Resp_Msg

TEXT
{Chck the status of children to make sure all are in completed state.}
Children_List := Get_Children(Proc_UID)
IF Children_List.Children(dom) <> 0 -->
   Check_Children_Status(Children_List.Children)

 [] OTHERWISE --> SKIP
FI

{Get a copy of PCB from the PRocess}
WHEN
    CP_To_Controller.Send(Action := Get_PCB) -->
    WHEN
        CP_To_Controller.Came -->
        Controller_Resp := CP_To_Controller.Get
    END
END
IF Controller_Resp.tag = t3 -->
   PCB := Controller_Resp.val
   {Get last Recovery point in order to generate lable for new
   RR}

   LRP:=Get_LRP(Proc_UID)
   LRP:=LRP+1
   L:=Assign_Label(Proc_UID,LRP)

   {CreateRP_Data_Record}
   RP_Data:=Create_RP_Data_Record(Proc_UID,PCB)

   {Get memory address & length of the segment for created RP_Data.}
    MM_Info:=Get_memory_addr(RP_Data)

   {Write the record on Stable Storage}
    SS_Resp := Write_To_SS(L , MM_Info)

   IF SS_Resp = ACK_Resp -->
      {update PM Data_base}
       PMDB_Resp:= Modify_DB(Proc_UID, LRP)
```

```
      Establish_Recovery_Point.val.OP::=ERP
      Establish_Recovery_Point.val.Param := LRP

    [] OTHERWISE  -->
      Establish_Recovery_Point.val := Undifined_Error
   FI

 [] OTHERWISE -->
   Establish_Recovery_Point.val := Undefined_Error
 FI

END Establish_Recovery_Point


CONTROLLER

  IF Work_Request.tag≠Establish_Recovery_Point -->
     Response.val::=Illegal_Command
   [] OTHERWISE -->
        Response::=Establish_Recovery_Point(Work_Request.val.Param)
  FI
 Parent_Iface.Send(Response)
 Terminate_Command_Processor

END  Establish_Recovery_Point
```

3.4.14 Machine DRP_Processor

```
DRP (My_Process_ID:Process_UID_Type,
     Work_Request:Appl_Req_Msg)

PUBLIC
      PMDB_Iface:Small_Mailbox(PMDB_Response_Type, PMDB_Request_Type)
      Parent_Iface:Large_Mailbox((Appl_Req_Msg, Response_Msg)
      SS_Iface:Port(SS_PM_Msg,PM_SS_Msg)


OBJECTS

      Timer_Interrupt IS Event_Rec INLET
      Timer_Iface:Small_Mailbox(Timer_Response,
                                Timer_Command)
      PM_Timer:Timer:=(Timer_Iface TO Timer_Command_Iface,
                       Timer_Interrupt TO Interrupt)

      Response : Response_Msg
```

3.4.14.1 Discard_RP

```
Discard_RP(Proc_UID : Process_UID_Type,RPnum1, RPnum2 : RP_Num)
RETURNS ACK

VARIABLES
      i : RP_Num
      L : Label_Type ARRAY
      j : INTEGER
      Resp : SS_PM_Msg
TEXT
    i := RPnum1
    j := 1
    DO i <= RPnum2 -->
       L(j) := Assign_Label(Proc_UID , i)
       i := i+1
       j := j+1
    OD

    WHEN
       SS_Iface.Send(<DRP, [Labels := s := L]>) -->
       WHEN
            SS_Iface.Came -->
             Resp := SS_Iface.Get
       END
    END
    IF Resp.val.Param = Success -->
       Discard_RP.OP := Discard_Recovery_Point
       Discard_RP.Param := NULL
END  Discard_RP


CONTROLLER

  IF Work_Request.tag≠DRP -->
     Response.val:=Illegal_Command
   [] OTHERWISE -->
        Response:=Discard_RP(Work_Request.val.Param)
  FI
 Parent_Iface.Send(Response)
 Terminate_Command_Processor

END

END  {Realization Dictionary}
```

3.5 SYSTEM Process_Manager

```
PUBLIC
    PM_TO_OS: Small_Mailbox (OS_PM_Msg,PM_OS_Msg)
    PM_TO_MM: Small_Mailbox (MM_PM_Msg,PM_MM_Msg)
    PM_TO_SS: Small_Mailbox (SS_PM_Msg,PM_SS_Msg)
    PM_TO_UIDgen: Small_Mailbox (UIDgen_PM_Msg,
                    PM_UIDgen_Msg)
    PM_TO_SS: Large_Mailbox(SS_PM_Msg,PM_SS_Msg)
    PM_TO_MM: Large_Mailbox(MM_PM_Msg,PM_MM_Msg)


OBJECTS

        PM_TO_Timer IS Event_Rec INLET
        Connection_to_Timer:Small_Mailbox(Timer_Response,
                                          Timer_Command)
        PM_Timer:Timer:=(Connection_to_Timer TO Timer_Command_Iface,
                         PM_TO_Timer TO Interrupt)
        SS_Port:Port_Multiplexer(SS_PM_Msg, PM_SS_Msg)
        MM_Port:Port_Multiplexer(MM_PM_Msg, PM_MM_Msg)

        PMDB:PM_Database_Manager:=(SS_Iface TO SS_Port.Iface,
                                   OS_Iface TO Router_To_PMDB)
        PM_TO_Process:  LIST(Large_Mailbox (Appl_Req_Msg,Response_Msg))
        PM_TO_Processor_Scheduler:LIST(Large_Mailbox(
                                  Processor_Scheduler_Req_Msg,Response_Msg))
        PM_TO_Router:Small_Mailbox (Router_PM_Msg,PM_Router_Msg)
        PM_TO_Router_Control_CMD: Small_Mailbox (Router_PM_CTRL
                                   _Msg,PM_Router,CTRL_Msg)
        Router: Router_Machine: = (Router_To_PM_Control_CMD
                                   TO PM_TO_Router_Control_CMD,
                                   Router_TO_PM TO PM_TO_Router,
                                   Router_TO_OS:= PM_TO_OS)

        Database_Iface: LIST (Large_Mailbox(PMDB_Request_Type,
                              PMDB_Response_Type))

        Command_Proc_Iface: LIST(Large_Mailbox(CP_to_Controller,
                                 Controller_to_CP))


        Appl_Process_Pool: Process POOL
        New_Proc_Indx: Process INDEX

            {Command Processors Pool}

        Delete_Command_Proc:Delete_Processor POOL
        Delete_CP:Delete_Processor INDEX
        Commit_Command_Proc:Commit_Processor POOL
        Commit_CP:Commit_Processor INDEX
```

```
Abort_Command_Proc:Abort_Processor POOL
Abort_CP:Abort_Processor INDEX
Rollback_Command_Proc:Rollback_Processor POOL
Rollback_CP:Rollback_Processor INDEX
ERP_Command_Proc:ERP_Processor POOL
ERP_CP:ERP_Processor INDEX
Create_Command_Proc:Create_Processor POOL
Create_CP:Create_Processor INDEX
DRP_Command_Proc:DRP_Processor POOL
DRP_CP:DRP_Processor INDEX
End_Trans_Command_Proc:End_Trans_Processor POOL
End_Trans_CP:End_Trans_Processor INDEX
```

3.5.1 Procedure Extend_Router_Mbx

```
Extend_Router_Mbx(kind:(Appl,Cmnd)) RETURNS INTEGER

VARIABLES
   To_Router:PM_Router_CTRL_CMD_Msg

TEXT

 {Send a request to router to create a new mailbox.}
IF kind=Appl --> To_Router.tag::=Create_Appl_Mailbox
   kind=Cmnd --> To_Router.tag::=Create_PM_Mbx
FI
 To_Router.val::=Proc_UID

 WHEN
     PM_TO_Router_CTRL_CMD.Send(To_Router) -->
     WHEN
         PM_TO_Router_CTRL_CMD.Came -->
         Extend_Router_Mbx := PM_TO_Router_CTRL_CMD.Get
     END
 END

{end of Extend_Router_Mbx)
```

3.5.2 Delete_Router_Mbx

```
Delete_Router_Mbx(kind:(Appl, Cmnd),
                        Proc_ID:Process_UID_Type)
RETURNS ACK

VARIABLES
   To_Router:PM_Router_CTRL_CMD_Msg

TEXT
```

```
 {Send a request to router to create a new mailbox.}
IF kind=Appl --> To_Router.tag::=Delete_Appl_Mailbox
   kind=Cmnd --> To_Router.tag::=Delete_PM_Mbx
FI
 To_Router.val::=Proc_UID

 WHEN
     PM_TO_Router_CTRL_CMD.Send(To_Router) -->
     WHEN
         PM_TO_Router_CTRL_CMD.Came -->
         Extend_Router_Mbx := PM_TO_Router_CTRL_CMD.Get
     END
 END

{end of Extend_Router_Mbx)
```

3.5.3 Procedure Invoke

```
 Invoke(Invoker, Receiver:Process_UID_Type,
                 Request:General_Msg_Type
                 MBX:Invoke_Iface) RETURNS ACK

VARIABLE
     To_Router:Invoke_Msg_Type
TEXT

To_Router.Object_UID ::=Receiver
To_Router.Caller_UID ::=Invoker
To_Router.Operation::=Request

    {Send the message to Router}
 WHEN
   MBX.Send(To_Router)-->
   WHEN
     MBX.came --> Invoke::=MBX.Get
   END
 END
{end of Invoke}
```

3.5.4 Procedure Extend_Mbx

```
 Extend_Mbx(Mbx_Array:LIST(T))
 RETURNS INTEGER

{This procedure extends the array representation of the LIST parameter
 and returns the hi-bound of the array}
```

TEXT

```
    i::=Extend_Mbx(Command_Proc_Iface) {Extends the mailbox array and return
                                        the high bound of the array}
    k::=Extend_Router_Mbx(Cmnd) {Extends Router_TO_PM mailbox list}
    Proc_ID::=Get_UID(process)
    Remote := FALSE

  IF CP=Delete -->
           Delete_CP::=Delete_Command_Proc.create((Proc_ID, Remote,
             Caller_UID, Work_Request),
             Parent_Iface TO Invoker_Iface,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

   {store the index Delete_CP with the following mailboxes:
    Command_Proc_Iface, Router_TO_PM, CP_TO_SS, CP_TO_MM}

   CP=Abort -->
           Abort_CP::=Abort_Command_Proc.create((Proc_ID, Remote,
             Caller_UID, Work_Request),
             Parent_Iface TO Invoker_Iface,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

   {store the index Abort_CP with the following mailboxes:
    Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

   CP=Commit -->
           Commit_CP::=Commit_Command_Proc.create((Proc_ID, Remote,
             Caller_UID, Work_Request),
             Parent_Iface TO Invoker_Iface,
             PM_Controller_Iface TO Command_Proc_Iface(i),
```

```
                PMDB_Iface TO PMDB.Database_Port,
                OS_Iface TO Router_TO_PM(k),
                SS_Iface TO SS_Port.Iface,
                MM_Iface TO MM_Port.Iface)

{store the index Commit_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=Create -->
             Create_CP::Create_Command_Proc.create((Proc_ID, Remote,
                Caller_UID, Work_Request),
                Parent_Iface TO Invoker_Iface,
                PM_Controller_Iface TO Command_Proc_Iface(i),
                PMDB_Iface TO PMDB.Database_Port,
                OS_Iface TO Router_TO_PM(k),
                SS_Iface TO SS_Port.Iface,.
                MM_Iface TO MM_Port.Iface)

{store the index Create_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

 CP=ERP -->
             ERP_CP::ERP_Command_Proc.create(Proc_ID(, Remote,
                Caller_UID, Work_Request),
                Parent_Iface TO Invoker_Iface,
                PM_Controller_Iface TO Command_Proc_Iface(i),
                PMDB_Iface TO PMDB.Database_Port,
                OS_Iface TO Router_TO_PM(k),
                SS_Iface TO SS_Port.Iface,
                MM_Iface TO MM_Port.Iface)

{store the index ERP_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=Rollback -->
             Rollback_CP::Rollback_Command_Proc.create((Proc_ID, Remote,
                Caller_UID, Work_Request),
                Parent_Iface TO Invoker_Iface,
                PM_Controller_Iface TO Command·Proc_Iface(i),
                PMDB_Iface TO PMDB.Database_Port,
                OS_Iface TO Router_TO_PM(k),
                SS_Iface TO SS_Port.Iface,
                MM_Iface TO MM_Port.Iface)

{store the index Rollback_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=End_Trans -->
            End_Trans_C.'::End_Trans_Command_Proc.create((Proc_ID, Remote,
                Caller_UID, Work_Request),
                Parent_Iface TO Invoker_Iface,
                PM_Controller_Iface TO Command_Proc_Iface(i),
                PMDB_Iface TO PMDB.Database_Port
```

```
                    OS_Iface TO Router_TO_PM(k),
                    SS_Iface TO SS_Port.Iface,
                    MM_Iface TO MM_Port.Iface)

      {store the index End_Trans_CP with the following mailboxes:
       Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

        CP=DRP_Command_Proc -->
              DRP_CP::=DRP_Command_Proc.create((Proc_ID, Remote,
                 Caller_UID, Work_Request),
                 Parent_Iface TO Invoker_Iface,
                 PM_Controller_Iface TO Command_Proc_Iface(i),
                 PMDB_Iface TO PMDB.Database_Port,
                 OS_Iface TO Router_TO_PM(k),
                 SS_Iface TC SS_Port.Iface,
                 MM_Iface TO MM_Port.Iface)

      {store the index DRP_CP with the following mailboxes:
       Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

END {of Create_New_Machine}
```

3.5.6 Procedure Create_Command_Processor

```
 Create_Command_Processor(New_Machine_Type:Command_Machine_Type,
                          Work_Request:Appl_Req_Msg,
                          New_Slot:Integer,
                          Invoker_Machine_Type:Command_Machine_Type,
                          Indx:Machine_Index_Type {for the Invoker machine})

TEXT

 IF Invoker_Machine_Type=Delete -->
      Create_New_Machine(New_Machine_Type,
                    Delete_Processor(Indx).Descendent_Iface(New_Slot),
                    Work_Request)

    Invoker_Machine_Type=Abort -->
       Create_New_Machine(New_Machine_Type,
                    Abort_Processor(Indx).Descendent_Iface(New_Slot),
                    Work_Request)

    Invoker_Machine_Type=Create -->
       Create_New_Machine(New_Machine_Type,
                    Create_Processor(Indx).Descendent_Iface(New_Slot),
                    Work_Request)
```

```
    Invoker_Machine_Type=Rollback -->
        Create_New_Machine(New_Machine_Type,
                           Rollback_Processor(Indx).Descendent_Iface(New_Slot),
                           Work_Request)

    Invoker_Machine_Type=Commit -->
        Create_New_Machine(New_Machine_Type,
                           Commit_Processor(Indx).Descendent_Iface(New_Slot),
                           Work_Request)

    Invoker_Machine_Type=ERP -->
        Create_New_Machine(New_Machine_Type,
                           ERP_Processor(Indx).Descendent_Iface(New_Slot),
                           Work_Request)

    Invoker_Machine_Type=DRP -->
        Create_New_Machine(New_Machine_Type,
                           DRP_Processor(Indx).Descendent_Iface(New_Slot),
                           Work_Request)

    Invoker_Machine_Type=End_Trans -->
        Create_New_Machine(New_Machine_Type,
                           End_Trans_Processor(Indx).Descendent_Iface(New_Slot),
                           Work_Request)
 FI


END {of Create_Command_Processor}
```

3.5.7 Procedure Create_Appl_Server

```
 Create_Appl_Server(CP:Command_Machine_Type,
          OBJ Invoker_Iface:Large_Mailbox(Response_Msg,
                                          Appl_Req_Msg),
          Work_Request:Appl_Req_Msg)

VARIABLES i,j,k,l,m:INTEGER
          Proc_ID:Process_UID_Type

TEXT

      i:=Extend_Mbx(Command_Proc_Iface) {Extends the mailbox array and return
                                         the high bound of the array}
      k:=Extend_Router_Mbx(Cmnd) {Extends Router_TO_PM mailbox list}
      Proc_ID:=Get_UID(process)


   IF CP=Delete -->
            Delete_CP:=Delete_Command_Proc.create(Proc_ID, Work_Request,
```

```
            Parent_Iface := Invoker_Iface,
            PM_Controller_Iface TO Command_Proc_Iface(i),
            PMDB_Iface TO PMDB.Database_Port,
            OS_Iface TO Router_TO_PM(k),
            SS_Iface TO SS_Port.Iface,
            MM_Iface TO MM_Port.Iface)

{store the index Delete_CP with the following mailboxes:
 Command_Proc_Iface, Router_TO_PM, CP_TO_SS, CP_TO_MM}

CP=Abort -->
         Abort_CP::Abort_Command_Proc.create(Proc_ID, Work_Request,
            Parent_Iface := Invoker_Iface,
            PM_Controller_Iface TO Command_Proc_Iface(i),
            PMDB_Iface TO PMDB.Database_Port,
            OS_Iface TO Router_TO_PM(k),
            SS_Iface TO SS_Port.Iface,
            MM_Iface TO MM_Port.Iface)

{store the index Abort_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

CP=Commit -->
         Commit_CP::Commit_Command_Proc.create(Proc_ID, Work_Request,
            Parent_Iface := Invoker_Iface,
            PM_Controller_Iface TO Command_Proc_Iface(i),
            PMDB_Iface TO PMDB.Database_Port,
            OS_Iface TO Router_TO_PM(k),
            SS_Iface TO SS_Port.Iface,
            MM_Iface TO MM_Port.Iface)

{store the index Commit_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=Create -->
         Create_CP::Create_Command_Proc.create(Proc_ID, Work_Request,
            Parent_Iface := Invoker_Iface,
            PM_Controller_Iface TO Command_Proc_Iface(i),
            PMDB_Iface TO PMDB.Database_Port,
            OS_Iface TO Router_TO_PM(k),
            SS_Iface TO SS_Port.Iface,
            MM_Iface TO MM_Port.Iface)

{store the index Create_CP with the following mailboxes:
 Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

 CP=ERP -->
         ERP_CP::ERP_Command_Proc.create(Proc_ID, Work_Request,
            Parent_Iface := Invoker_Iface,
            PM_Controller_Iface TO Command_Proc_Iface(i),
            PMDB_Iface TO PMDB.Database_Port,
            OS_Iface TO Router_TO_PM(k),
```

```
            SS_Iface TO SS_Port.Iface,
            MM_Iface TO MM_Port.Iface)

    {store the index ERP_CP with the following mailboxes:
     Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

     CP=Rollback -->
            Rollback_CP::=Rollback_Command_Proc.create(Proc_ID, Work_Request
              Parent_Iface := Invoker_Iface,
              PM_Controller_Iface TO Command_Proc_Iface(i),
              PMDB_Iface TO PMDB.Database_Port,
              OS_Iface TO Router_TO_PM(k),
              SS_Iface TO SS_Port.Iface,
              MM_Iface TO MM_Port.Iface)

    {store the index Rollback_CP with the following mailboxes:
     Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

     CP=End_Trans -->
           End_Trans_CP::=End_Trans_Command_Proc.create(Proc_ID, Work_Reques
              Parent_Iface := Invoker_Iface,
              PM_Controller_Iface TO Command_Proc_Iface(i),
              PMDB_Iface TO PMDB.Database_Port
              OS_Iface TO Router_TO_PM(k),
              SS_Iface TO SS_Port.Iface,
              MM_Iface TO MM_Port.Iface)

    {store the index End_Trans_CP with the following mailboxes:
     Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

     CP=End_Trans -->
           DRP_CP::=DRP_Command_Proc.create(Proc_ID, Work_Request,
              Parent_Iface := Invoker_Iface,
              PM_Controller_Iface TO Command_Proc_Iface(i),
              PMDB_Iface TO PMDB.Database_Port,
              OS_Iface TO Router_TO_PM(k),
              SS_Iface TO SS_Port.Iface,
              MM_Iface TO MM_Port.Iface)

    {store the index DRP_CP with the following mailboxes:
      Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

END {of Create_Appl_Server}
```

3.5.8 Procedure Create_Remote_Appl_Server

```
 Create_Remote_Appl_Server(CP:Command_Machine_Type,
          Work_Request:Appl_Req_Msg)

VARIABLES i,j,k,l,m:INTEGER
          Proc_ID:Process_UID_Type
```

TEXT

```
    i::=Extend_Mbx(Command_Proc_Iface) {Extends the mailbox array and return
                                        the high bound of the array}
    k::=Extend_Router_Mbx(Cmnd) {Extends Router_TO_PM mailbox list}
    Proc_ID::=Get_UID(process)


IF CP=Delete -->
           Delete_CP::=Delete_Command_Proc.create(Proc_ID, Work_Request,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

  {store the index Delete_CP with the following mailboxes:
   Command_Proc_Iface, Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=Abort -->
           Abort_CP::=Abort_Command_Proc.create(Proc_ID, Work_Request,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

  {store the index Abort_CP with the following mailboxes:
   Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

  CP=Commit -->
           Commit_CP::=Commit_Command_Proc.create(Proc_ID, Work_Request,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

  {store the index Commit_CP with the following mailboxes:
   Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

   CP=Create -->
           Create_CP::=Create_Command_Proc.create(Proc_ID, Work_Request,
             PM_Controller_Iface TO Command_Proc_Iface(i),
             PMDB_Iface TO PMDB.Database_Port,
             OS_Iface TO Router_TO_PM(k),
             SS_Iface TO SS_Port.Iface,
             MM_Iface TO MM_Port.Iface)

  {store the index Create_CP with the following mailboxes:
   Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}
```

```
      CP=ERP -->
             ERP_CP::=ERP_Command_Proc.create(Proc_ID, Work_Request,
               PM_Controller_Iface TO Command_Proc_Iface(i),
               PMDB_Iface TO PMDB.Database_Port,
               OS_Iface TO Router_TO_PM(k),
               SS_Iface TO SS_Port.Iface,
               MM_Iface TO MM_Port.Iface)

     {store the index ERP_CP with the following mailboxes:
      Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

       CP=Rollback -->
             Rollback_CP::=Rollback_Command_Proc.create(Proc_ID, Work_Request
               PM_Controller_Iface TO Command_Proc_Iface(i),
               PMDB_Iface TO PMDB.Database_Port,
               OS_Iface TO Router_TO_PM(k),
               SS_Iface TO SS_Port.Iface,
               MM_Iface TO MM_Port.Iface)

     {store the index Rollback_CP with the following mailboxes:
      Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

       CP=End_Trans -->
            End_Trans_CP::=End_Trans_Command_Proc.create(Proc_ID, Work_Reques
               PM_Controller_Iface TO Command_Proc_Iface(i),
               PMDB_Iface TO PMDB.Database_Port
               OS_Iface TO Router_TO_PM(k),
               SS_Iface TO SS_Port.Iface,
               MM_Iface TO MM_Port.Iface)

     {store the index End_Trans_CP with the following mailboxes:
      Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

       CP=End_Trans -->
            DRP_CP::=DRP_Command_Proc.create(Proc_ID, Work_Request,
               PM_Controller_Iface TO Command_Proc_Iface(i),
               PMDB_Iface TO PMDB.Database_Port,
               OS_Iface TO Router_TO_PM(k),
               SS_Iface TO SS_Port.Iface,
               MM_Iface TO MM_Port.Iface)

     {store the index DRP_CP with the following mailboxes:
       Command_Proc_Iface,  Router_TO_PM, CP_TO_SS, CP_TO_MM}

END {of Create_Appl_Server}
```

3.5.9 Procedure Destroy_Command_Processor

```
 Destroy_Command_Proc(Command_Proc:Command_Machine_Type,
          Indx:Machine_Index_Type,
          Proc_UID :Process_UID_Type)
```

TEXT

{This procedure destroys the designated command processor machine. All of its mailbox connections are destroyed and the storage space occupied by these mailboxes is released}

3.5.10 CONTROLLER

```
VARIABLES
        Msg : Appl_Req_Msg
        Caller_UID : Process_UID_Type
        Response : Response_Msg
        TO_Process : Response_Msg
        Resp : ACK
        i,j : INTEGER

WHENEVER

{check the application command mailboxes to see if there is any message}
(i:0..PM_To_Process.Hi_Bound) PM_TO_Process.Element(i).came -->

   Msg:=PM_TO_Process.Element(i).Get
   Caller_UID :=PM_TO_Process.Element(i).Get_UID
   {check the msg type}
   IF (Msg.val.OP=Create_Process) OR (Msg.val.OP=Create_Transaction) OR
      (Msg.val.OP=Begin_Transaction) -->
        Create_Appl_Server(Create, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP=Delete_Process -->
        Create_Appl_Server(Delete, PM_TO_Process.Element(i), Msg)


     [] Msg.val.OP = Establish_Recovery_Point OR ERP -->
        Create_Appl_Server(ERP, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP = Rollback -->
        Create_Appl_Server(Rollback, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP=Discard_Recovery_Point -->
        Create_Appl_Server(DRP, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP = End_Transaction -->
        Create_Appl_Server(End_Trans, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP = Commit -->
        Create_Appl_Server(Commit, PM_TO_Process.Element(i), Msg)

     [] Msg.val.OP = Status_ Query -->
        PM_TO_Database.Send(OS_Msg)
        WHEN
          PM_TO_Database.Came --> Response:=PM_TO_Database.Get
```

```
            PM_TO_Process.Element(i).Send(Response)
        END

   FI



{Check the Command Processor Interfaces for any request message}
[] (i:Command_Proc_Iface.lob..Command_Proc_Iface.hib)
Command_Proc_Iface.Element(i).Came -->
     Rqt:=Command_Proc_Iface.Element(i).Get
     Caller_Machine_Indx::=Command_Proc_Iface.Element(i).Get_Index

   IF Rqt.tag=Create -->
      Create_Command_Processor(Rqt.val.Machine_Type,
         Rqt.val.Work_Request, Rqt.val.Descendent_Mbx_Indx,
         Rqt.val.Caller_Machine_Type, Caller_Machine_Indx)

      Rqt.tag=Destroy -->
      Destroy_Command_Processor(Caller_Machine_Type, Caller_Machine_Index,
         Proc_UID)

      Rqt.tag=Service_Call -->

         IF Rqt.val.Action=New_Process -->
              {Create a new process and return its UID to the caller}
              J::=Extend_Router_Mbx(Appl)
              K::=Extend(PM_TO_Process) {Returns the index of the new element
              L::=Extend(PM_TO_Process_Scheduler)
              Proc_UID ::=Get_UID(Process)

              New_Proc_Indx::=Appl_Process_Pool.Create
                 (Process_TO_PM TO PM_TO_Process(K),
                  Process_TO_Scheduler TO PM_TO_Process_Scheduler(L),
                  Process_TO_Router TO Router_TO_Process(J))

              {Store this index and the Process UID in appropriate
              Large Mailboxes}

              Command_Proc_Iface.Element(i).Send(Proc_UID)

           [] Rqt.val.Action=Destroy_Process -->
              Process_Indx::=Find_Process_indx(Rqt.val.UID))
              Appl_Process_Pool.Destroy(Process_Indx)

             {This will also release all connections to the destroyed proces

              Command_Proc_Iface.Element(i).Send(Success)

           [] Rqt.val.Action=Run_Process -->
               PCB::=Rqt.val.Param,
               K::=PM_TO_Proc_Scheduler.Get_Index(PCB.PUID)
```

```
        WHEN
          PM_TO_Database.Came --> Response::PM_TO_Database.Get
              PM_TO_OS.Send(Receiver::Caller_UID, Sender::PM_UID,
                            Operation::Response])
        END
      FI

    FI

FI


END {Process Manager}
```

```
            WHEN
              PM_TO_Proc_Scheduler.Element(K).Send(OP::=Run_Process,
                  Param::=PCB) --> SKIP
            END
            Command_Proc_Iface.Element(i).Send(Success)

       [] Rqt.val.Action=Stop_Process -->
            Proc_UID ::=Rqt.val.Param
            K::=PM_TO_Proc_Scheduler.Get_Index(Proc_UID)
            WHEN
              PM_TO_Proc_Scheduler.Element(K).Send(OP::=Stop_Process)
              ---> WHEN
                     PM_TO_Proc_Scheduler.Element(K).Came -->
                     PCB::=PM_TO_Proc_Scheduler.Element(K).Get.Param
                   END
            END

              Command_Proc_Iface(i).Send(PCB)

      FI

{Check the Operation Switch interface for request/response messages}
[] PM_TO_OS.Came -->
     OS_Msg::=PM_TO_OS.Get
     Caller_UID ::=OS_Msg.Sender
     Remote := TRUE

   IF OS_Msg.Operation.tag=Appl_Req_Msg -->
     IF (OS_Msg.Operation.val.OP=Create_Process) OR
        (OS_Msg.Operation.val.OP=Create_Transaction) OR
        (OS_Msg.Operation.val.OP=Begin_Transaction) -->
           Create_Remote_Appl_Server(Create,Remote, Caller_UID, Msg)

     [] OS_Msg.Operation.val.OP=Delete_Process -->
        Create_Remote_Appl_Server(Delete, Remote, Caller_UID, Msg)


     [] OS_Msg.Operation.val.OP = Establish_Recovery_Point OR ERP -->
        Create_Remote_Appl_Server(ERP, Remote, Caller_UID, Msg)

     [] OS_Msg.Operation.val.OP = Rollback -->
        Create_Remote_Appl_Server(Rollback, Remote, Caller_UID, Msg)

     [] OS_Msg.Operation.val.OP=Discard_Recovery_Point -->
        Create_Remote_Appl_Server(DRP, Remote, Caller_UID, Msg)

     [] OS_Msg.Operation.val.OP = Commit -->
        Create_Remote_Appl_Server(Commit, Remote, Caller_UID, Msg)

     [] OS_Msg.Operation.val.OP = Status_Query -->
        PM_TO_Database.Send(OS_Msg)
```

CHAPTER 4

TYPE MANAGER DESIGN

This chapter presents the design of the generic object manager in the Ze system in CSDL.

Section 4.1, Machines Dictionary, describes the interfaces and behavior various machines used in the design definiton of the Type Manager syste These machine defintions do not contain the details of the internal structu of the machines.Section 4.2 contains the type definitions for the vario object types used in the entire design definition. Section 4.3 defin various procedures that are used by several machines in the desi definitions. The details of the SYSTEM Type_Manager architectures is given section 4.4 titled Realization Dictionary.

4.1 MACHINE DICTIONARY

```
SYSTEM Type_Manager(T:TYPE)

PUBLIC

       TM_To_PM:Small_Mailbox(PM_TM_Msg_TM_PM_Msg)
       TM_To_Router:Small_Mailbox(Router_TM_Msg,TM_Router_Msg)
       Servers_To_PM:List(small_Mailbox(PM_Server_Msg,
                                        Server_TM_Msg))
       Servers_To_PM_PS:List(Small_Mailbox(PM_Server_PS_Msg,
                                           Server_PM_PS_Msg))

BEHAVIOR
     { This machine accepts operation request from the
       TM_To_Router; Participate in commit protocols
       with PMs through TM_To_Router.}

END  Type_Manager


Timer
     {The machine is imported from Proc/Trans management (Page 4-17)}
```

4.2 TYPES DICTIONARY

```
Imported from Process/Transaction manager:
   1) Small_Mailbox
   2) List
   3) NULL
   4) Transaction_Status_Type
   5) UID_Type

  Lock_Mode_Type IS (Read, Update, Unlock)

  Object_Status IS (Uncommitted, Commit_Pending, Committed, Aborted)

  Transaction_Context_List IS Transaction_UID_Type ARRAY

  Object_Version_Type(T:TYPE) IS
  MODEL [Obj_Version_No:INTEGER,
         Version_Status:Object_Status,
         Creator_ID:Transaction_Context_List,
         Time_Stamp:INTEGER,
         Previous_Version:Object_Version_Type,
         Object:T]

  Let %Obj_Version:Object_Version_Type

  End Object_Version_Type

  Operation_Name IS ABSTRACT

  {Each Type has specific operation set. A typical
   operation set could be defined as follow.}
  Operation_Type IS [Op1, [op:operation_name,
                           Ty: (Read,Update),
                           Param:Parameters] []
                    Op2, ...............] UNION

  { This type is the union of several types.}
  TM_To_Router_Msg_Type IS ABSTRACT

  Invoke_Param_Type IS [Object_UID:UID_Type,
                        Client_UID:Transaction_Context_List,
                        Initial_Access:BOOLEAN,
                        Time_Stamp:INTEGER,
                        Operation_Info:Operation_Type]

  Msg_Type IS (PREPARE, COMMIT, ABORT, COMPLETED)

  Msg_Param_Type IS [Obj_ID : UID_Type,
                    Client_ID : Transaction_Context_List]

  Rollback_Param_Type IS [Obj_ID : UID_Type,
                          Transaction : Transaction_Context_List,
```

```
                              Children : UID_Type ARRAY,
                              Time_Stamp : INTEGER]

Abort_Param_Type IS [Obj_ID : UID_Type,
                     Transaction : Transaction_Context_List,
                     Children : UID_Type ARRAY]

Request_Type IS [Invoke, Invoke_Param_Type []
                 T1, [Op : Msg_Type,
                      Params : Msg_Param_Type] []
                 Rollback, Rollback_Param_Type []
                 Abort , Abort_Param_Type] UNION

Invoke_Response_Msg IS (Done, Abort, Queued)

De_Q_Msg IS [Exist,Invoke_Param_Type  []
             Not_Exist, NULL] UNION

Info_Rec IS [Flag : BOOLEAN,
             UID  : Transaction_Context_List]

Object_Header(T : TYPE) IS [object_UID:UID_Type,
                            IS_Waiting:BOOLEAN,
                            Current_Version:Object_Version_Type(T),
                            Obj_Status:Object_Status,
                            Lock_Mode:Lock_Mode_Type,
                            Current_Client_ID:Complete_TCL ARRAY]
```

4.2.1 Definition of abstract data type for Complete TCL

```
Complete_TCL IS
MODEL [ ID : Transactin_Context_List,
        Mode : Lock_Mode_Type,
        Converted : BOOLEAN,

Let %TCL:Complete_TCL

OFUN Remove_Leaf_Transaction

PRE TRUE
POST %TCL'.dom = %TCL.dom - 1


BEHAVIOR
     {This function removes the leaf transaction from
      the transaction context list on which this function
      is applied.}

END Complete_TCL
```

4.2.2 Definition of abstract data type for Queue

```
Queue_Type IS
MODEL [Invoke_Param_Type ARRAY]
LET %Q: Queue_Type

OFUN Enqueue (Req:Invoke_Param_Type)
PRE TRUE
POST %Q.high = Req
BEHAVIOR
      {It adds the request to the queue}

OFUN Dequeue (Object_UID : UID_Type,
              Mode : Lock_Mode) RETURNS De_Q_Msg
BEHAVIOR
      {It removes the request, which is waiting for the
       object with the given uid and mode, from the queue.}

END Queue_Type
```

4.2.3 Definition of abstract data type for Set_of_Objects

```
Set_of_Objects (T:TYPE) IS
MODEL [[Header:Object_Header(T),
        Version:Object_Version_Type (T) ARRAY ] ARRAY ]

LET %Objects:Set_of_Objects

OFUN Assign_Header (H:Object_Header)
PRE TRUE
POST %Objects.high.Header = H
BEHAVIOR
   {It adds the Header of new created objects to the object
    list.}

OFUN Attach_New_Version(Obj_ID:UID_Type,
                        V:Object_Version_Type)

PRE TRUE
POST %Objects' (i).Version.high=V
     Where i:%Objects.lob < i < %Objects.hib
           AND %Objects (i).Header.Object_UID=Obj_ID)

BEHAVIOR
    {This function adds the new version of the object
     with given obj-ID to the end of the version list.}

OFUN Force_Header (Obj_ID:UID_Type)RETURNS BOOLEAN
PRE TRUE
POST
BEHAVIOR
```

```
        {This function sends a request to Stable Storage manager
         to write out the header of the object with given obj_ID
         on stable storage.}

OFUN Force_Version (Obj_ID:UID_Type) RETURNS BOOLEAN
PRE TRUE
POST
BEHAVIOR
        {This function sends a request to stable storage manager
         to write out the latest version of the object with
         given obj_ID on stable storage.}

VFUN Get_Lock_Mode (Obj_ID:UID-Type)
     RETURNS Lock_Mode_Type

PRE TRUE
POST Get_Lock_Mode =%Objects (i).Header.Lock_Mode
     Where i:%Objects.lob< i < %Object_ID = Obj.ID

BEHAVIOR
      {It returns the lock mode for the
       requested object.}


OFUN Add_Client(Obj_ID:UID_Type,
                Client_TCL:Complete_TCL)

PRE TRUE
POST %Objects (i).Header.Current_Client_ID(hib) = Client_Tch
       Where i :%Objects.lob<i<%Objects.hib
                AND %Objects (i).Header.object_ID=Obj_ID

BEHAVIOR
      {It adds a new client to the set of current
       clients of given object.}


VFUN  Obj_Status (Obj_ID:UID_Type)
      RETURNS Object_Status

BEHAVIOR
      {It returns the object status of the specified
       object.}


OFUN  Converted(Obj_ID : UID_Type,
                Client_ID : Transaction_Context_List)
      RETURNS BOOLEAN
BEHAVIOR
     {This function returns TRUE if the lock held by
      given client was converted from the update lock
      of its ancestor.}
```

```
OFUN  Change_Status (Obj_ID:UID_Type,
                     Obj_Status:Object_Status)

PRE TRUE
POST  %Objects(i).Header.Obj_Status = Obj_Status
      Where i: %Objects.lob<i<%Objects.hib
            AND %Objects(i).Header.Object_ID=Obj_ID

BEHAVIOR
      {This function modifies the Status of
       Given function.}


OFUN  Change_Lock_Mode (Obj_ID:UID_Type,
                        LM:Lock_Mode:Type)

PRE TRUE
POST  %Objects (i).Header.Lock_Mode= LM
      Where i:%Objects.lob<i<%Objects.Hib
          AND %Objects(i).Header.Object_ID=Obj_ID

BEHAVIOR
     {This function changes the lock mode of
      specified object to given mode.  In order to
      release the lock, we should change the mode
      to unloc}


VFUN  Is_Locked_By_Ancestor(Obj_ID : UID_Type,
                            Client_ID : Transaction_Context_List)
      RETURNS Info_Rec
BEHAVIOR
     { This function checks to see whether any
       ancestor of the given client's holds a
       lock on the specified object or not.  }

VFUN Is_Lock_Holder(Obj_ID : UID_Type,
                    Client_UID : Transaction_Context_List)
     RETURNS BOOLEAN

BEHAVIOR
     { This function checks whether the specified client
       is currently holding a lock on the given object
       or not.}

VFUN  More_Readers(Obj_ID : UID_Type,
                   Client_ID : Transaction_Context_List)
     RETURNS BOOLEAN

BEHAVIOR
     { This function returns true if there is more than
       one transaction holding a read lock on the given
       object.}
```

```
VFUN Deadlock_Prev_Alg(Obj_ID : UID_Type,
                       Param : Transaction_Context_List)
     RETURNS BOOLEAN

BEHAVIOR
     { This function performs a deadlock prevention
       algorithm, namely "Wound_Wait" algorithm, in
       order to determine whether the transactin
       to be aborted or queued.}

OFUN  Delete_Version (Obj_ID : UID_Type,
                      IDs : Transaction_UID_Type ARRAY,
                      Time_Stamp : INTEGER)
BEHAVIOR
     { This function deletes all the object versions created
       after specified Time_Stamp by the given transaction(s).
       Also it modifies the current version field in object
       header to point to the correct copy.}

OFUN  Create_New_Version (Obj_ID : UID_Type,
                          Creator : Transaction_Context_List,
                          Status : Object_Status)
      RETURNS Object_Version_Type

BEHAVIOR
     { This function creates a new version of the given
       object and adds it to its object version list.
       The given status is the status of newly created
       version.}

OFUN  Change_Waiting(Obj_ID : UID_Type)
PRE TRUE
POST %Objects'(i).Header.Is_Waiting := NOT (%Objects(i).Header.Is_Waitin
     Where i:%Objects.lob<i<%Objects.Hib
          AND %Objects(i).Header.Object_ID=Obj_ID
BEHAVIOR
     {This function negates the value of the Is_Waiting
      field (which indicates whether any transaction is
      waiting to acquire a lock on that object or not)
      in object header. }

OFUN  Is_Waiting (Obj_ID : UID_Type)
      RETURNS BOOLEAN
PRE  TRUE
POST Is_Waiting := %Objects(i).Header._Waiting
BEHAVIOR
     {This function returns the value of the Is_Waiting
      field of the object header.}

OFUN  Change_Client_Mode(Obj_ID : UID_Type,
                         Mode : Lock_Mode_Type)
```

```
PRE  TRUE
POST %Objects(i).Header.Current_Client_ID.high.Mode = Mode

OFUN Release_Lock(Obj_ID : UID_Type,
                  Client_ID : Transaction_Context_List)
BEHAVIOR
     { This function releases the lock which is held
       by the leaf transaction of given client and
       returns the lock to its parent.
       This function consist of the following steps:
         - Lock for the client_ID in the current client
           list of the object header.
         - Check whether the TCL of the client transaction's
           parent exists in the Current_Client_List
         - If it does, determine what mode the parent
           inherits the object in and modify the parent's
           mode if neccessary and remove the client_ID
           from the current client list.
         - If the parent TCL does not exist, simply remove
           the leaf transaction from the client's TCL  and
           leave the new TCL in current client list.


       The locking Mode which the parent inherits the
       object in is determined by the following table:
```

Lock released by child	Lock previously held by parent	Lock currently held by parent
Read	Read	Read
Read	Update	Update
Read	None	Read
Update	Read	Update
Update	Update	Update
Update	None	Update

```
OFUN  Check_Path(Obj_ID : UID_Type,
                 Client_ID : Transaction_Context_List)
      RETURNS BOOLEAN
BEHAVIOR
     {This function returns True if no transaction on the
      path from the current lock holder to the successor of
      the least common upper bound of the holder and given
      client has inherited an update lock.}

OFUN  Check_Dependency(Obj_ID : UID_Type,
                       UIDs : Transaction_UID_Type ARRAY)
      RETURNS BOOLEAN
BEHAVIOR
     {This function returns True if there exist a version
      of the given object in commit_Pending state, created
```

```
        after the object was modified by any of the transactions
        in the given UID list and the creator of the version
        is not among those in the list.}

  OFUN  Convert_Lock(Obj_ID : UID_Type,
                     Client_ID : Transaction_Context_List)
  BEHAVIOR
       {This function changes the lock mode of the object
        header and the given Client to update mode.
        This function is used whenever a transaction which
        already holds a read lock requests for an update lock.}

  OFUN  Commit_Prev_Version(Obj_ID : UID_Type,
                            Client_ID : Transasction_Context_List)
  BEHAVIOR
       { This function commits the previous Commit_Pending
         version of given object which was created either
         by the transaction itself or by any of its children.
         This function is usually applied to those objects
         which are in uncommitted or aborted state and a
         Commit message is addressed to them.}

  OFUN  Restore_Prev_Version( Obj_ID : UID_Type)
  BEHAVIOR
       { This function restore the previous Committed version
         of the given object.}

  OFUN  Apply (Obj_ID:UID_Type,
               OP_name:Operation_Type)

  BEHAVIOR
        {For the time being, it is assummed that
         this function performs the requested
         operation on the latest version of the
         object with given Obj_ID. A possible
         implementation is a pool of servers.}

  END  Set_of_Objects


END   TYPES DICTIONARY
```

4.3 PROCEDURE DICTIONARY

4.2.3.1 PROCEDURE Invoke_Proc

```
PROCEDURE Invoke_Proc (Params:Invoke_Param_Type)

BEHAVIOR
     { This procedure will invoke Lock_Grant procedure
       to perform lock granting algorithm before it
       performs the requested operation. Then, depends
       on the result of the lock granting algorithm
       it will do the corresponding actions.}

VARIABLES
        Resp:Invoke_Response_Msg
        Response:TM_To_Router_Msg_type

TEXT
        Resp::=Lock_Grant (Params)

        IF Resp=Grant-->
           Objects.Apply (Params.Obj_ID,Params.Op.name)
           Response.val.Msg::=Done
           Response.val.Param::=Params.
           TM_To_Router.Send (Response)

        [] Resp=Abort -->
           Response.val.Msg::=Abort
           Response.val.Param::=Params
           TM_To_Router.Send (Response)

        [] OTHERWISE --> SKIP {when a request has been queued}

        FI

END {Invoke_Proc}
```

4.2.3.2 PROCEDURE Lock_Grant

```
PROCEDURE Lock_Grant (Params:Invoke_Params_Type)
RETURNS Invoke_Response_Msg

BEHAVIOR
     { This procedure is executed whenever an object is
       accessed to carry out the lock granting algorithm.
       The following is the list of all possible lock modes
```

which an object can be in and thier corresponding actions.}

CONDITION	Requested lock	ACTION
Initial Access:		
1)Object unlock	Read/Update	Grant
2)Read lock by T granted	Read	No action
by OM	Update	Grant if no more readers
		Queue otherwise
3)Read lock by ancestor of T	Read	Grant if Rule1
converted from Update lock	Update	Grant if Rule2
of its ancestor		Queue otherwise
4)Read lock by ancestor of T	Read	Grant
granted by OM	Update	Grant if no more readers
		Grant otherwise
5)Update lock by ancestor of T	Read/Update	Grant
6)Read lock by non_ancestor of	Read	Grant
T granted by OM	Update	Queue
7)Read lock by non_ancestor of		
T converted from update lock	Read/Update	Queue
of its ancestor		
8)Update lock by non_ancestor	Read/Update	Queue
of T		
Subsequent request:		
1)Object unlock	Read/Update	Abort
2)Object not locked by T	Read/Update	Abort
3)Read lock by T granted by OM	Read	No action
	Update	Grant if no more reader
		Queue otherwise
4)Read lock by T converted from	Read	No action
update lock of its ancestor	Update	Grant
5)Update lock by T	Read/Update	No action

Rule1 : No transaction on the path from the current lock holder to the successor of the least common upper bound of the holder and requester has inherited an update lock

Rule2 : Rule1 and no other transaction is currently holding a read lock

VARIABLES

Lock_Mode:Lock_Mode_Type
Ancestor : Info_Rec
OK:BOOLEAN
Lock_Holder:BOOLEAN
More_Readers:BOOLEAN
Requested_Lock : Lock_Mode_Type
CTCL : Complete_TCL

TEXT

```
{check the Lock Mode of the object.}
Lock_Mode::=Objects.Get_Lock_Mode (Params.Obj_ID)
Requested_Lock := Params.Operation_Info.val.Ty
Ancestor := Objects.Is_Locked_by_Ancestor(Params.Obj_ID,
                                          Params.Client_ID)
IF  Params.Initial_Access -->
    IF Lock_Mode = Unlock -->
       {grant the requested lock & update the object.}
       CTCL := Generate_Complete_TCL(Params.Client_ID,
                                    Requested_Lock FFALSE)
       Objects.Update_Object(Params.Obj_ID, CTCL)
       Lock_Grant::=Grant

     [] IF (Lock_Holder) AND (Lock_Mode = Read) -->
          IF Requested_Mode = Update -->
             IF NOT Objects.More_Readers -->
                Objects.Convert_Lock(Params.Obj_ID)
                Lock_Grant := Grant

              [] OTHERWISE --> {More readers }
                Queue_Request (Param)
             FI

           [] OTHERWISE --> SKIP
          FI
     [] Ancestor.Flag -->
       IF Requested_Lock = Read -->
          IF Objects.Converted(Params.Obj_ID,
                               Ancestor.UID) -->
             IF Objects.Check_Path -->
                CTCL := Generate_Complete_TCL(Params.Client_ID,
                                              Requested_Lock, TRUE)
                Objects.Update_Object(Param.Obj_ID, CTCL)
                Lock_Grant := Grant
              [] OTHERWISE --> Queue_Request(Params)
             FI
          [] OTHERWISE -->
             IF (Requested_Lock = Read) OR
                ((Requested_Lock=Update) AND
                 (NOT Objects.More_Readers)) -->
                 CTCL := Generate_Complete_TCL(Params.Client_ID,
                                              Requested_Lock, TRUE)
                 Objects.Update_Object(Param.Obj_ID, CTCL)
                 Lock_Grant := Grant
              [] OTHERWISE --> Queue_Request(Params)
             FI
        [] OTHERWISE --> { lock_mode = update }
             CTCL := Generate_Complete_TCL(Params.Client_ID,
                                           Requested_Lock, TRUE)
             Objects.Update_Object(Param.Obj_ID, CTCL)
             Lock_Grant := Grant
       FI
    [] OTHERWISE --> { locked by non ancestor }
```

```
        IF (NOT Objects.Converted(Param.Obj_ID)
            AND Requested_Mode = Read) -->
            CTCL := Generate_Complete_TCL(Params.Client_ID,
                                          Requested_Lock, TRUE)
            Objects.Update_Object(Param.Obj_ID, CTCL)
            Lock_Grant := Grant
         ▯ OTHERWISE --> Queue_Request(Params)
        FI
    FI
    ▯  OTHERWISE --> {It is not an initial access.}
            Lock_holder := IS_Lock_holder (Params.Obj_ID,
                                           Params.Client_UID)
            IF (Lock_Holder) AND (Lock_Mode = Read) AND
               (Params.Operation_Info.val.Ty=Update) -->
                More_Reads:= Objects.More_Readers (Params.Obj_ID,
                                                   Params.Client_ID)
                IF More_Readers -->
                  {There are other Transactions which are holding Read
                   lock on this object}
                   OK:=Deadlock_Prev_Alg (Params.Obj_ID, Params.Client_ID
                   IF OK --> {enqueue the request}
                      IF Objects.Is_Waiting(Param.Obj_ID) --> SKIP

                       ▯ OTHERWISE -->
                         Objects.Change_Waiting(Param.Obj_ID)
                      FI
                      queue.Enqueue (Params)
                   ▯ OTHERWISE -->
                      Lock_Grant::=Abort
                   FI
                 ▯ OTHERWISE --> {convert the Lock_Mode}
                     Convert_Lock (Params)
                     Lock_Grant::=Grant

                 FI
            ▯  (NOT (lock_Holder) OR Lock_Mode = Unlock) -->
                    Lock_Grant ::=Abort
            ▯  OTHERWISE --> SKIP

        FI

    FI

END  Lock_Grant
```

4.2.3.3 PROCEDURE Prepare_Proc

```
PROCEDURE Prepare_Proc (Param:Msg_Param_Type)
```

BEHAVIOR

```
    { Upon recieving a PREPARE message, Object Manager will
      invoke this procedure to perform the corresponding
      actions according to the following table.

              CONDITION                    ACTION
             -----------                 ---------
   1) Read lock by T and status     Send READY msg
      is Committed                  Unlock the object
   2) Update lock by T and
    a) Status = Uncommitted         Change status to Commit_Pending
                                    Force object on SS
                                    Send READY msg
    b) Status = Commi_Pending       Send READY msg
    c) Status = Committed           No action
   3)  All other conditions         Send ABORT msg
```

VARIABLES

```
         Lock_Holder:BOOLEAN
         Mode:Object_Mode_Type
         Status:Object_Status
```

TEXT

```
    Mode::=Object.Get_Object_Mode (Param.Obj_ID)
    Lock_Holder::=Objects.Is_Lock_Holder (Param.Obj_ID,Param.Client_ID)
    Status::=Objects.Obj_Status (Param.Obj_ID)
    Response.val.Param := Param
       IF  Lock_Holder --> {The transaction is currently holding a lock}
         IF Mode = READ -->
            IF Status = Committed -->
               Objects.Release_Lock (Param.Obj_ID, Param.Client_ID
              Response.val.Msg := READY
              TM_To_Router.Send (Response)
            [] OTHERWISE -->
              Response.val.Msg:= ABORT
              TM_To_Router.Send (Response)
            FI
        [] Mode = Update -->
            IF Status = Uncommitted -->
               Objects.Change_Status (Param.Obj_ID, Commit_Pending)
              Objects.Force_Version (Param.Obj_ID)
              Response.val.Msg := READY
              TM_To_Router.Send (Response)
            [] Status = Commit_Pending -->
              Response.val.Msg := READY
              TM_To_Router.Send (Response)
            [] Status = Abort -->
               Objects.Restore_Prev_Version (Param.Obj_ID)
              Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
```

```
               Response.val.Msg := ABORT
               TM_To_Router.Send (Response)
            [] OTHERWISE --> Skip {Status = Committed}
         FI
    FI
   [] OTHERWISE --> {does not hold any lock}
      Response.val.Msg := ABORT
      TM_To_Router.Send (Response)

END Prepare_Proc
```

4.2.3.4 PROCEDURE Completed_Proc

```
PROCEDURE Completed_Proc (Param:Msg_Param_Type)

BEHAVIOR
     { This procedure is invoked when a COMPLETED message
       is received from PM. Depends on the status of the
       object, proper actions will be taken.
       In the following table all the possible conditions
       are listed.}
```

CONDITIONS	ACTIONS
1) object is locked by T or a descendents of T in Read mode & status=Committed	Unlock the object & send an ACK message
2) object is locked by T or a descendents of T in Update mode &	
a) status = Committed	Unlock the object & send an ACK message
b) status = Commit_Pending	same as condition 2a plus change status to Committed & force the object on SS
c) status = Aborted or Uncommitted	Restore previos version & unlock the object & send an ACK message
3) object is locked by a stranger	send an ACK message

* The conditions which are not listed in the above table are those which should never arise in a reliable system.

```
VARIABLES
        Lock_Holder: BOOLEAN
        Mode:Object_Mode_Type
        Status:Object_Status
```

```
        Response:TM_To_Router_Msg_Type
        Descendent: BOOLEAN

TEXT

        Mode::=Objects.Get_Object_Mode(Param.Obj_ID)
        Lock_Holder:-Objects.Is_Lock_Holder(Param.Obj_ID,
                                             Param.Client_ID)
        Status::=Objects.Obj_Status (Param.Obj_ID)
        Response.val.Msg::=ACK
        Response.val.Param::=Param
        Descendent::=Objects.Is_Lock_by_Descendent (Param.Obj_ID,
                                                    Param.Client_ID)
        IF   Mode = Unlock -->
             TM_To_Router.Send (Response)

             [] Mode = Read -->
                IF (Lock_Holder OR Descendent) AND Status = Committed -->
                    Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
                    TM_To_Router.Send (Response)

             [] OTHERWISE -->
                    TM_To_Router.Send (Response)

             FI

        []   Mode = Update -->
             IF (lock_Holder OR Descendent) -->
                IF Status = Committed
                   Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
                   TM_To_Router.Send (Response)

             [] Status = Commit_Pending -->
                Objects.Change_Status (Param.Obj_ID,
                                       Param.Client_ID)
                Object.Force_Version (Param.Obj_ID)
                TM_To_Router.Send (Response)

             [] (Status = Uncommitted OR Status = Aborted)-->
                Objects.Restore_Prev_Version (Param.Obj_ID)
                Objects.Force_Header (Param.Obj_ID)
                Objects.Release_Lock (Param.Obj_ID,Param.Client_ID)
                TM_To_Router.Send (Response)

             FI

         [] OTHERWISE -->
               TM_To_Router.Send (Response)

         FI

    FI
```

```
END  Completed_Proc
```

4.2.3.5 PROCEDURE Time_Out_Proc

```
PROCEDURE Time_Out_Proc (Param: Time_Out_Param_Type)

BEHAVIOR
     { This procedure is executed whenever a timeout
       interrupt is received from Timer.
       In the following table all the possible conditions
       and the actions needed to be taken are listed.

            CONDITIONS              ACTIONS
           ------------            ---------
   1)Object is locked in       unlock the object
     Read mode
   2)Object is locked in
     Update mode &
     a)status = Uncommitted/   Restore the previous
                Aborted        Committed version
                               unlock the object
                               Force the object on SS
     b)status = Commit_Pending  Send a query to the PM regarding
                                 the status of Client transaction,
                               if transaction is non existant then
                               abort the version and restore the
                               previous committed version of the
                               object otherwise set the timer
                               again.

VARIABLES
            Mode:Object_Mode_Type
            Status: Object_Status
            Response:TM_To_Router_Msg_Type
            Resp:Transaction_Status_Type

TEXT

       Mode::=Objects.Get_Object_Mode(Param.Obj_ID, Param.Client_ID)
       Status::=Objects.Obj_Status (Param.Obj_ID)

       IF   Mode = Read -->
            IF Status = Committed -->
               Objects.Release_Lock (Param.Obj_ID, Param. Client_ID)

            [] OTHERWISE --> SKIP
            FI
```

```
        [] Mode = Update -->
             IF (Status = Uncommitted OR Status = Aborted ) -->
                Objects.Restore_Prev_Version (Param.Obj_ID)
                Objects.Force_Header (Param.Obj_ID)
                Objects.Release_Lock (Param.Obj_ID,Param.Client_ID)

           [] Status = Commit_Pending -->
                  {send status query for client-Transaction
                   and wait until you get a response back
                  IF Resp = Non_Existent
                  THEN
                     Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
                     Objects.Restore_Prev_Version (Param.Obj_ID)
                  ELSE
                     set the timer.
                  }

           [] OTHERWISE --> SKIP

          FI

     FI

END Time_Out_Proc
```

4.2.3.6 PROCEDURE Commit_Proc

```
PROCEDURE Commit_Proc (Param:Msg_Param_Type)

BEHAVIOR
     {Upon recieving a Commit message from PM
      Object manager perfoems the corresponding
      actions according to the lock mode and
      status of the object.
      The following is the list of all possible
      conditions and thier corresponding actions.}
```

CONDITIONS	ACTIONS
1) Object is locked by T in Read mode & Status = Committed	Unlock the object Force the object header Send an ACK message
2) Object is locked by T in Update mode & Status=Commit_Pending	Unlock the object Change status to committed Force Object header Send an ACK message
3) Object is locked by some descendent of T &	
a) Status = Commit_Pending/ Committed	The same as condition 2

```
   b) Status = Uncommitted/       Commit the previous
               Aborted            Commit_Pending version
                                  Unlock the object
                                  Send an ACK message
4) Object is unlocked or
   locked by some stranger        Send ACK message
```

* The conditions which are not listed in the above table are those which should never arise in a reliable system.

VARIABLES

```
Lock_Holder:  BOOLEAN
Mode:Object_Mode_Type
Status:Object_Status
Response:TM_To_Router_Msg_Type
Ancestor : BOOLEAN
```

TEXT

```
Mode::=Objects.Get_Object_Mode (Param.Obj_ID)
Lock_Holder::=Object.Is_Lock_Holder (Param.Obj_ID,
                                     Param.Client_ID)
Status::=Objects.Obj_Status(Param.Obj_ID)
Ancestor::=Objects.Is_Lock_by_Ancestor (Param.Obj_ID,
                                        Param.Client_ID)

Response.val.Msg::=ACK
Response.val.Param::=Param

IF Lock_Holder -->
   IF (Mode = Read AND Status = Committed) -->
      Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
      TM_To_Router.Send (Response)

   [] (Mode = Update AND Status = Commit_Pending) -->
      Objects.Change_Status (Param.Object_ID,
                             Committed)

      Object.Force_Version (Param.Object_ID)
      Objects.Release_Lock (Param.Obj_ID, Param. Client_ID)
      TM_To_Router.Send (Response)

   [] OTHERWISE --> SKIP

   FI

[] OTHERWISE --> {the transaction does not hold any lock
                  itself}
```

```
            IF  Objects.Is_Lock_by_Descendent (Param.Obj_ID,
                                               Param.Client_ID) -->
                IF (Mode = Read AND Status = Committed) -->
                   Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
                   TM_To_Router.Send (Response)

                [] OTHERWISE --> SKIP

            FI

        [] Mode = Update -->
            IF STATUS = Commit_Pending -->
               Objects.Change_Status (Param.Obj_ID,
                                      Committed)

               Objects.Force_Version (Param.Obj_ID)
               Objects.Release_Lock (Param.Obj_ID, Param.Client_ID)
               TM_To_Router.Send (Response)

            [] (Status = Uncommitted OR
                Status = Aborted) -->
                Objects.Commit_Prev_Version(Param.Obj_ID, Param.Client_ID)
                Objects.Release_Lock(Param.Obj_ID,Param.Client_ID)
                Objects.Force_Version(Param.Obj_ID)
                Objects.Force_Header(Param.Obj_ID)
                TM_To_Router.Send(Response)

            [] (Status = Committed) -->
                Objects.Release_Lock (Param.Obj_ID,Param.Client_ID)
                Objects.Force_Header(Param.Obj_ID)
                TM_To_Router.Send (Response)

         FI

    [] OTHERWISE --> {Since there is no info about this transaction in
                      OM database. Just send an ACK back.}

        TM_To_Router.Send (Response)

    FI

 FI

END Commit_Proc.
```

4.2.3.7 PROCEDURE Rollback_Proc

```
PROCEDURE Rollback_Proc( Param : Rollback_Param_Type)
```

BEHAVIOR

{ This procedure checks whether the given transaction is currently holding an update lock on the object, if it does then this procedure discards all the versions created by that transaction with time stamp larger than the time stamp of the recovery point. Also, if any of the transaction children have updated the object, then it discards all the versions created by those transactions. The following is the list of all possible conditions and thier corresponding actions:

CONDITIONS	ACTIONS
1) Object is unlock	Send an Abort message.
2) Object is locked by a stranger.	Send an Abort message.
3) Object is locked by T in Read mode	Send an ACK message
4) Object is locked by T in Update mode	Delete all versions created by the transaction children. Delete all versions created by the transaction with TS > recovery point TS. Force modified object header on stable storage. Send an ACK message

* The conditions which are not listed in the above table are those which should never arise in a reliable system.

VARIABLES

```
        Mode : Lock_Mode_Type
        Descendent : BOOLEAN
        Lock_Holder : BOOLEAN
        Response : TM_To_Router_Msg_Type
```

TEXT

```
        Mode := Object.Get_Lock_Mode(Param.Obj_ID)
        Lock_Holder := Objects.Is_LOck_Holder(Param.Obj_ID,
                                              Param.Transaction)
        IF NOT Lock_Holder -->
           Descendent := Object.Is_Locked_By_Descendent(Param.Obj_ID,
                                                        Param.Transaction)
         [] OTHERWISE --> SKIP
        FI
        Response.val.Param := Param
```

```
        IF Mode = Unlock -->
           Response.val.Msg := Abort
           TM_To_Router.Send(Response)

         [] OTHERWISE -->  {The object is lock }
           IF (NOT Lock_Holder) AND (NOT Descendent) -->
              Response.val.Msg := Abort
              TM_To_Router.Send(Response)

            [] OTHERWISE --> { The object is lock either by the
                                    client or one of its children.}
              IF Mode = Read -->
                 Objects.Release_Lock(Param.Obj_ID, Param.Transaction)
                 {Return the lock to its parent.}
                 Objects.Force_Header(Param.Obj_ID)
                 Response.val.Msg := ACK
                 TM_To_Router.Send(Response)

              [] Mode = Update -->
                 Objects.Delete_Version(Param.Obj_ID, Param.Children, -1)
                 Objects.Delete_Version(Param.Obj_ID, Param.Transaction,
                                        Param.Time_Stamp)
                 Objects.Release_Lock(Param.Obj_ID, Param.Transaction)
                 {Return to the parent.}
                 Objects.Force_Header(Param.Obj_ID)
              FI
        FI
   FI

END  Rollback_Proc
```

4.2.3.8 PROCEDURE Abort_Proc

```
 PROCEDURE Abort_Proc (Param : Abort_Param_Type)

BEHAVIOR
     {This procedure is executed whenever an Abort message
      is received from PM .
      In the following table all the possible conditions
      and the actions needed to be taken are listed .}
```

CONDITIONS	ACTIONS
1)Object is unlocked	Send an ACK message
2)Object is locked by T in Read mode & Object_Status=Committed	Unlock the object & give it to its parent Force the object header Send an ACK message
3)Object is locked by T in	

Update mode & a)Object_Status=Committed	Send an ACK message
b)Object_Status=Uncommitted/ Commit_Pending	IF Rule1 then Send an NACK message OTHERWISE: Delete all the versions created by this transaction and its children; Unlock the object & give it to its parent; Force object header; Send an ACK message
4)Object is locked by some other transaction except T	Send an ACK message

Rule1: If there exist a version of the given object in commit_Pending state, created after the object was modified by either the client transaction or any of the transactions in its attached list, and the creator of the version is not among those.

* conditions which are not listed in the above table are those which should never arise in a reliable system.

```
VARIABLES
        Lock_Holder : BOOLEAN
        Mode : Object_Mode_Type
        Status : Object_Status
        Response : TM_To_Router_Msg_Type
TEXT
        Mode := Objects.Get_Object_Mode(Param.Obj_ID)
        Lock_Holder := objects.Is_Lock_Holder(Param.Obj_ID,
                                              Param.Transaction)
        Status := Objects.Obj_Status(Param.Obj_ID)
        Response.val.Msg := ACK
        Response.val.Param := Param

        IF (Mode = Unlock) OR (NOT Lock_Holder) -->
          TM_To_Router.Send (Response)

         [] OTHERWISE -->   {object is locked by T}
          IF Mode=Read -->
             IF Status = Committed -->
                Objects.Release_Lock(Param.Obj_ID, Param.Transaction)
                Objects.Force_Header(Param.Obj_ID)
                TM_To_Router.Send(Response)
              [] OTHERWISE --> SKIP
                {This condition must never arise.}
             FI
           [] OTHERWISE --> {Object is locked in Update mode}
```

```
            IF ((Status = Uncommitted) OR
               (Status = Commit_Pending)) -->
               IF (NOT Objects.Check_Dependency(Param.Obj_ID,
                                                Param.Children)) -->
                 Objects.Delete_Version(Param.Obj_ID,Param.Transaction)
                 Objects.Delete_Version(Param.Obj_ID,Param.Children)
                 Objects.Release_Lock(Param.Obj_ID,Param.Transaction)
                 Objects.Change_Status(Param.Obj_ID,Committed)
                 Objects.Force_Header(Param.Obj_ID)
                 TM_To_Router.Send(Response)

                [] OTHERWISE -->  {Because of dependency the abort
                                        request should be refused.}
                 Response.val.Msg := NACK
                 TM_To_Router.Send(Response)
              FI

             [] OTHERWISE --> { Status = Committed}
                 TM_To_Router.Send(Response)
           FI

END  Abort_Proc
```

4.2.3.9 PROCEDURE Generate_Complete_TCL

```
PROCEDURE Generate_Complete_TCL(TCL : Transaction_Context_List,
                                LM : Lock_Mode_Type,
                                Converted : BOOLEAN)
RETURNS Complete_TCL

BEHAVIOR
     {This procedure receives a transaction context list
      ,the lock mode in which the object is to be locked
      and a boolean variable which indicates whether the
      lock was converted  from an update lock of its ancestor
      or not, and returns a complete TCL which contains all
      these information.}

END Generate_Complete_TCL
```

4.2.3.10 PROCEDURE Queue_Reaquest

```
PROCEDURE Queue_Request(Params : Invoke_Param_Type)

VARIABLES
      OK : BOOLEAN
```

```
TEXT
    {queue the request if it is older}
    OK:=Deadlock_Prev_Alg (Params.Object_ID,
                           Params.client_TCL)
    IF OK -->   {engueue the request}
       IF Objects.Is_Waiting(Param.Obj_ID) --> SKIP

        [] OTHERWISE -->
          Objects.Change_Waiting(Param.Obj_ID)
       FI
       queue.Enqueue (Params)
     [] OTHERWISE -->
        Lock_Grant::=Abort
    FI

END  Queue_Request
```

4.4 REALIZATION DICTIONARY

4.4.1 SYSTEM Type_Manager

```
SYSTEM Type_Manager(T:Type)
PUBLIC
      TM_To_PM:Small_Mailbox(PM_TM_Msg,TM_PM_Msg)
      TM_To_Router:Small_Mailbox(Router_TM_Msg,Tm_Router_Msg)
      Servers_To_PM:List(Small_Mailbox (PM_To_Servers,
                                        Servers_To_PM))
      Servers_To_PM_PS:List(Small_Mailbox(PM_Servers_PS_Msg,
                                        Servers_PM_PS_Msg))

BEHAVIOR

     {The Type_Manager communicates with PM and Router thru
      TM_To_PM and TM_To_Router interfaces respectively.
      It accepts operation requests through TM_To_Router
      interface and participates in commit protocols with
      PMs through TM_To_Router.}

OBJECTS

      Objects:Set_Of_Objects(T)
      Queue:Queue_Type
      TM_To_Timer IS Event_Rec INLET
      Connection_To_Timer:Small-Mailbox(Timer_Response,
                                        Timer_Command)
      Local_Timer:Timer:=(Connection_To_Timer TO
```

```
                          Timer_Command_Iface,
                          TM_To_Timer TO Interrupt)
    Server_Process:Process POOL
    Server_Index:Process INDEX
    Req : Request_Type
```

4.4.2 CONTROLLER

```
CONTROLLER

    WHENEVER
      TM_To_Router.Came -->
        Req:=TM_To_Router.Get
        IF Req.tag = Invoke --> Invoke_Proc(Req.val)
         [] Req.tag = T1 -->
          IF Req.val.Op = Prepare --> Prepare_Proc(Req.val.Param)
           [] Req.val.Op = Commit --> Commit_Proc(Req.val.Param)
           [] Req.val.Op = Completed --> Completed_Proc(Req.val.Param)
           [] OTHERWISE --> Error
          FI
         [] Req.tag = Abort --> Abort_Proc(Req.val)
         [] Req.tag = Rollback --> Rollback_proc(Req.val)
         [] OTHERWISE --> Error
        FI

      []TM_To_Timer.Came -->
        Req := TM_To_Timer.Get
        IF Req.tag = Time_Out  --> Time_Out_Proc(Req.val)

         [] OTHERWISE --> SKIP
        FI
    END

END Type_Manager.
```

Chapter 5

SYMBOLIC NAME MANAGER DESIGN

This chapter presents a formal definition of the detailed design of Symbolic Name Manager written in the Department of Defense' ADA language.

5.1 SNTM_Interface Packge Specification

```
PACKAGE SNTM_interface IS

  -- Create a brand new context object and return its uid in
  -- context_id.  The rel_class is a pre-defined set which
  -- determines on which host or hosts copies of the object are
  -- to be created.
  PROCEDURE create_context (rel_class: IN reliability_class;
                               status: OUT ret_stat);

  -- Delete a context object
  PROCEDURE delete_context (context_id: IN kernel.xtnded_uid;
                                status: OUT ret_status);

  -- Add a name-uid pair to an existing context object
  PROCEDURE add_name (context_id: IN kernel.xtnded_uid;
                            name: IN symb_name;
                         name_id: IN kernel.xtnded_uid;
                          status: OUT ret_stat);

  -- Remove a symbolic name from a context object
  PROCEDURE remove_name (context_id: IN kernel.xtnded_uid;
                               name: IN symb_name;
                             status: OUT ret_stat);

  -- Find a symbolic name in a context and return its associated
  -- uid in name_id.
  PROCEDURE lookup (context_id: IN kernel.xtnded_uid;
                          name: IN symb_name;
                       name_id: OUT kernel.xtnded_uid;
                        status: OUT ret_stat);
```

```
END SNTM_interface;
```

5.2 SNTM_Interface Package Body

```
PACKAGE BODY SNTM_interface IS

------------------
-- Create a brand new context object and return its uid in
-- context_id.  The rel_class is a pre-defined set which
-- determines on which host or hosts copies of the object are
-- to be created.
PROCEDURE create_context (rel_class: IN reliability_class;
                             status: OUT ret_stat) IS
  packed_parms : bit_string;

  BEGIN
  -- Pack the IN parameters for the kernel
  support.pack.create(rel_class, packed_parms);
  -- Call the kernel to transmit the request
  kernel.make_call(packed_parms);
  -- Call the kernel again to wait for the response
  kernel.get_response(packed_parms);
  -- Now unpack the result parameter(s)
  support.unpack.create(packed_parms, status);

  END create_context;

------------------
-- Delete a context object
PROCEDURE delete_context (context_id: IN kernel.xtnded_uid;
                              status: OUT ret_stat) IS
  packed_parms : bit_string;

  BEGIN
  -- Pack the IN parameters for the kernel
  support.pack.delete(context_id, packed_parms);
  -- Call the kernel to transmit the request
  kernel.make_call(packed_parms);
  -- Call the kernel again to wait for the response
  kernel.get_response(packed_parms);
  -- Now unpack the result parameter(s)
  support.unpack.delete(packed_parms, status);

  END delete_context;

-----------------
-- Add a name-uid pair to an existing context object
```

```
PROCEDURE add_name (context_id: IN kernel.xtnded_uid;
                          name: IN symb_name;
                       name_id: IN kernel.xtnded_uid;
                        status: OUT ret_stat) IS
  packed_parms : bit_string;

  BEGIN
  -- Pack the IN parameters for the kernel
  support.pack.add(context_id, name, name_id, packed_parms);
  -- Call the kernel to transmit the request
  kernel.make_call(packed_parms);
  -- Call the kernel again to wait for the response
  kernel.get_response(packed_parms);
  -- Now unpack the result parameter(s)
  support.unpack.add(packed_parms, status);


  END add_name;

-----------------
-- Remove a symbolic name from a context object
PROCEDURE remove_name (context_id: IN kernel.xtnded_uid;
                             name: IN symb_name;
                           status: OUT ret_stat) IS
  packed_parms : bit_string;

  BEGIN
  -- Pack the IN parameters for the kernel
  support.pack.remove(context_id, name, packed_parms);
  -- Call the kernel to transmit the request
  kernel.make_call(packed_parms);
  -- Call the kernel again to wait for the response
  kernel.get_response(packed_parms);
  -- Now unpack the result parameter(s)
  support.unpack.remove(packed_parms, status);


  END remove_name;


-----------------
-- Find a symbolic name in a context and return its associated
-- uid in name_id.
PROCEDURE lookup (context_id: IN kernel.xtnded_uid;
                        name: IN symb_name;
                     name_id: OUT kernel.xtnded_uid;
                      status: OUT ret_stat) IS

  packed_parms : bit_string;

  BEGIN
  -- Pack the IN parameters for the kernel
```

```
  support.pack.lookup(context_id, name, packed_parms);
  -- Call the kernel to transmit the request
  kernel.make_call(packed_parms);
  -- Call the kernel again to wait for the response
  kernel.get_response(packed_parms);
  -- Now unpack the result parameter(s)
  support.unpack.lookup(packed_parms, name_id, status);


  END lookup;

END SNTM_interface;
```

5.3 STNM Specification

```
PACKAGE SNTM IS

  WITH kernel;
  WITH TRANMGR;

  -- Declare the access pointer types to the operation tasks
  TYPE create_task IS LIMITED PRIVATE;
  TYPE delete_task IS LIMITED PRIVATE;
  TYPE add_task    IS LIMITED PRIVATE;
  TYPE remove_task IS LIMITED PRIVATE;
  TYPE lookup_task IS LIMITED PRIVATE;

  -- Declare the reliability class type
  TYPE reliability_class IS PRIVATE;

  -- Declare the return status type
  TYPE ret_stat IS (OK, LOCKED, NOT_FOUND, ERROR);

  -- Declare a symbolic name type
  TYPE symb_name IS STRING (1..max_name_len);

  -- Declare the operation type and task number used for
  -- identifying the operation tasks
  TYPE op_type IS (CREATE, DELETE, ADD, REMOVE, LOOKUP);
  TYPE op_task_num IS PRIVATE;
```

5.4 Controller Task Specification

```
-- Declare the type manager controller task. This task accepts
-- all incoming calls and allocates an operation task to
-- perform the actual operation.
TASK controller IS

   ENTRY create (rel_class: IN reliability_class;
                context_id: IN kernel.xtnded_uid;
                      orig: IN BOOLEAN;
                   task_id: OUT create_task);

   ENTRY delete (context_id: IN kernel.xtnded_uid;
                       orig: IN BOOLEAN;
                    task_id: OUT delete_task);

   ENTRY add (context_id: IN kernel.xtnded_uid;
                    name: IN symb_name;
                 name_id: IN kernel.xtnded_uid;
                    orig: IN BOOLEAN;
                 task_id: OUT add_task);

   ENTRY remove (context_id: IN kernel.xtnded_uid;
                       name: IN symb_name;
                       orig: IN BOOLEAN;
                    task_id: OUT remove_task);

   ENTRY lookup (context_id: IN kernel.xtnded_uid;
                       name: IN symb_name;
                       orig: IN BOOLEAN;
                    task_id: OUT lookup_task);

   -- This entry is used by the operation tasks to notify the
   -- controller that the operation is complete
   ENTRY op_done (operation: op_type;
                     op_num: op_task_num);

END controller;



PRIVATE        -- Private declarations for the SNTM package

 SUBTYPE reliability_class IS INTEGER RANGE 1..num_rel_class;
 SUBTYPE op_task_num IS INTEGER RANGE 1..num_tasks;

 ---------------- OPERATION TASK SPECIFICATIONS ----------------
 -- Declare task types for each of the operation tasks so the
```

```
-- controller can actually create an array of each from which
-- he will allocate tasks to actually perform the operations
----------------
TASK TYPE create_cntxt IS
   -- This entry is used by the controller to initialize the
   -- operation task and to pass it its unique task number
   ENTRY init (my_num: IN op_task_num);

   -- This entry is called by the controller to startup the
   -- operation task and pass it the required parameters
   ENTRY start (rel_class: IN reliability_class;
               context_id: IN kernel.xtnded_uid;
                     orig: IN BOOLEAN);

   -- This entry is called by the user's interface package to
   -- obtain the return values of the operation or the error
   -- status if it was unsuccessful
   ENTRY done (context_id: OUT kernel.xtnded_uid;
                   status: OUT ret_stat);
END create_cntxt;
----------------
TASK TYPE delete_cntxt IS
   ENTRY init (my_num: IN op_task_num);
   ENTRY start (context_id: IN kernel.xtnded_uid;
                      orig: IN BOOLEAN);
   ENTRY done (status: OUT ret_stat);
END delete_cntxt;
---------------
TASK TYPE add_name IS
   ENTRY init (my_num: IN op_task_num);
   ENTRY start (context_id: IN kernel.xtnded_uid;
                      name: IN symb_name;
                   name_id: IN kernel.xtnded_uid;
                      orig: IN BOOLEAN);
   ENTRY done (status: OUT ret_stat);
END add_name;
---------------
TASK TYPE remove_name IS
   ENTRY init (my_num: IN op_task_num);
   ENTRY start (context_id: IN kernel.xtnded_uid;
                      name: IN symb_name;
                      orig: IN BOOLEAN);
   ENTRY done (status: OUT ret_stat);
END remove_name;
---------------
TASK TYPE lookup_name IS
   ENTRY init (my_num: IN op_task_num);
   ENTRY start (context_id: IN kernel.xtnded_uid;
                      name: IN symb_name;
                      orig: IN BOOLEAN);
   ENTRY done (name_id: OUT kernel.xtnded_uid;
                status: OUT ret_stat);
END lookup_name;
```

```
------------ ACCESS VARIABLES TO OPERATION TASKS --------------
-- Declare access pointers to each of the operation tasks
TYPE create_task IS ACCESS create_cntxt;
TYPE delete_task IS ACCESS delete_cntxt;
TYPE add_task    IS ACCESS add_name;
TYPE remove_task IS ACCESS remove_name;
TYPE lookup_task IS ACCESS lookup_name;

-------------------- REPLICATION DATA RECORD ------------------
-- Declare a replication data record which contains all the
-- data necessary to distribute the copies or representatives
-- of a new context object
TYPE rep_data IS
   RECORD
      n: NATURAL;     -- Number of votes for this copy
      w: NATURAL;     -- Write quorum
      r: NATURAL;     -- Read quorum
      v: NATURAL;     -- Version (not used in rel. classes)
      h: ARRAY (POSITIVE RANGE <>) OF  -- Host list
            RECORD
                  id: kernel.xtnded_uid;     -- Host uid
               votes: NATURAL;               -- # of votes
            END RECORD;
   END RECORD;

------------------- CONTEXT OBJECT TYPE DECLARATION -----------
-- Define the types of locks on a context object
TYPE lock_type IS (READ, WRITE, NONE);
-- Define the internal structure of a context type object
TYPE context IS
   RECORD
      header: rep_data;       -- Object's replication data
        lock: lock_type;      -- Type of lock pending
        data: big_hash_table; -- Name->UID mappings
   END RECORD;

END SNTM;     -- End of type manager package specification
```

5.5 SNTM Package Body

```
PACKAGE BODY SNTM IS

  ----------------- CONTROLLER TASK BODY -----------------------
  TASK BODY controller IS
```

```
-- Declare the allocation list for each of the operation
-- task types.
----------------
create_pool: ARRAY (1..num_tasks) OF
   RECORD
      free: BOOLEAN;        -- Is this task available?
      task: create_task;    -- Access pointer to the task
   END RECORD;
----------------
delete_pool: ARRAY (1..num_tasks) OF
   RECORD
      free: BOOLEAN;
      task: delete_task;
   END RECORD;
----------------
add_pool: ARRAY (1..num_tasks) OF
   RECORD
      free: BOOLEAN;
      task: add_task;
   END RECORD;
----------------
remove_pool: ARRAY (1..num_tasks) OF
   RECORD
      free: BOOLEAN;
      task: remove_task;
   END RECORD;
----------------
lookup_pool: ARRAY (1..num_tasks) OF
   RECORD
      free: BOOLEAN;
      task: lookup_task;
   END RECORD;


--------------- ALLOCATION LIST FUNCTIONS -----------------
-- These functions handle the allocation lists defined above
-- They are used only within the controller and thus are
-- defined within its body and are totally private

FUNCTION get_task(op: IN op_type) RETURN op_task_num IS
BEGIN
   -- Find a free task for the given operation
   -- Mark its allocation list to show it is now busy
   -- Return the number of the task used
END get_task;

PROCEDURE free_task(op: IN op_type;
                   num: IN op_task_num) IS
BEGIN
   -- Mark the task numbered num in the given allocation
   -- list as being free again
END free_task;
```

```
BEGIN          -- Controller body code
   -- Create the new operation tasks and initialize the
   -- allocation lists with their access pointers.  Then
   -- mark the appropriate entry in the list to show the
   -- new task is free and call its initialization entry
   FOR i IN 1..num_tasks LOOP
      create_pool(i).task := NEW create_cntxt;  -- Create task
      create_pool(i).free := TRUE;              -- Show it is free
      create_pool(i).task.init(i);              -- Its number is i
      delete_pool(i).task := NEW delete_cntxt;
      delete_pool(i).free := TRUE;
      delete_pool(i).task.init(i);
      add_pool(i).task := NEW add_name;
      add_pool(i).free := TRUE;
      add_pool(i).task.init(i);
      remove_pool(i).task := NEW remove_name;
      remove_pool(i).free := TRUE;
      remove_pool(i).task.init(i);
      lookup_pool(i).task := NEW lookup_name;
      lookup_pool(i).free := TRUE;
      lookup_pool(i).task.init(i);
   END LOOP;

   -- Now loop waiting for operation requests and done calls
   -- from the completed operation tasks
   LOOP SELECT
      ACCEPT create (rel_class: IN reliability_class;
                    context_id: IN kernel.xtnded_uid;
                          orig: IN BOOLEAN;
                       task_id: OUT create_task) DO
         task_id := get_task(CREATE);       -- Allocate a task
         task_id.start(rel_class, orig);    -- Start it up
      END create;
   OR
      ACCEPT delete (context_id: IN kernel.xtnded_uid;
                           orig: IN BOOLEAN;
                        task_id: OUT delete_task) DO
         task_id := get_task(DELETE);
         task_id.start(context_id, orig);
      END delete;
   OR
      ACCEPT add (context_id: IN kernel.xtnded_uid;
                        name: IN symb_name;
                     name_id: IN kernel.xtnded_uid;
                        orig: IN BOOLEAN;
                     task_id: OUT add_task) DO
         task_id := get_task(ADD);
         task_id.start(context_id, name, name_id, orig);
      END add;
   OR
      ACCEPT remove (context_id: IN kernel.xtnded_uid;
                           name: IN symb_name;
```

```
                               orig: IN BOOLEAN
                           task_id: OUT remove_task) DO
         task_id := get_task(REMOVE);
         task_id.start(context_id, name, orig);
      END remove;
   OR
      ACCEPT lookup (context_id: IN kernel.xtnded_uid;
                           name: IN symb_name;
                           orig: IN BOOLEAN;
                        task_id: OUT lookup_task) DO
         task_id := get_task(LOOKUP);
         task_id.start(context_id, name, orig);
      END lookup;
   OR
      ACCEPT op_done (operation: IN op_type;
                         op_num: IN op_task_num) DO
         free_task(operation, op_num); -- Free the task
      END op_done;

   END SELECT;
   END LOOP;

END controller;   -- End of controller body




--------------------- CREATE_CNTXT TASK BODY ------------------
TASK BODY create_cntxt IS
   my_num: op_task_num;    -- Our task # in the controller
   host: kernel.xtnded_uid;   -- Storage for host id
   new_context: kernel.xtnded_uid;  -- New context UID
   status: ret_stat;        -- Operation status

   -- The reliability classes in this array are presumed to
   -- be pre-defined when the type manager is installed so
   -- that a user creating a new context need only select one
   -- of the available choices of configuration of copies
   -- for the object
   rep_class: ARRAY (reliability_class) OF rep_data;
   ---------------
   -- Define a local function to create a context instance
   -- on the local host
   FUNCTION local_create RETURN ret_stat IS
   BEGIN
      -- Call the kernel to allocate storage for a local
      -- copy of a context object instance
      -- Return status of OK if successful or ERROR if not
   END local_create;
   ---------------
   -- Define a local function to send a create request to a
   --    remote host and get his response
```

```
   FUNCTION remote_create(which_host: IN kernel.xtnded_uid)
      RETURN ret_stat;
   BEGIN
      -- Call a support package to pack the new context UID
      --    and the other parameters
      -- Call the kernel to send the call to the host ident-
      --    ified in the which_host parameter
      -- Wait for the response from the remote host
      -- Return the status received from the remote call
   END remote_create;


   END remote_create;


BEGIN       -- Create operation task body
   -- Accept the initialization call from the controller
   -- and remember our task number in my_num
   ACCEPT init (my_num: IN op_task_num) DO
   END init;

   LOOP     -- One operation at a time forever
      -- Accept the startup call from the controller and get
      -- the input parameters for the call
      ACCEPT start (rel_class: IN reliability_class;
                   context_id: IN kernel.xtnded_uid;
                         orig: IN BOOLEAN) DO
      END Start;  -- Release the controller from the rendevouz

      -- If this is the original call, we must act as
      -- collector for the replication algorithm
      IF orig THEN
         -- Get a fresh UID from the kernel for this context
         kernel.get_uid(context, new_context);

         -- Start a transaction so we can be sure that ALL of
         -- the copies get created or NONE of them do
         TRANMGR.begin_transaction;

         -- Loop through the host list of the given reliability
         -- class and do a remote call to each of those hosts
         host := support.next_host(rep_class(rel_class));
         LOOP
            IF host = my_host THEN  -- If our host is on list
               status := local_create; -- Create a local copy
            ELSE                     -- For each remote host
               status := remote_create(host);   -- Send it out
            END IF;
            host := support.next_host(rel_class); -- Next host

            -- Exit loop when list is exhausted or the local
            -- or a remote invocation fails for some reason
```

```
            EXIT WHEN (host = NULL_UID) OR (status /= OK);
         END LOOP;

         -- The create operation is a special one because it
         -- requires that ALL the requested copies be success-
         -- fully created (not just a write quorum). Therefore
         -- the transaction is aborted if any of the subordi-
         -- nates failed.
         IF status /= OK THEN TRANMGR.Abort_transaction;

         -- End the transaction (either commit or abort)
         TRANMGR.end_transaction;

      ELSE     -- If this is a subordinate call, just do it
         new_context := context_id;
         status := local_create;

      END IF;

      -- Accept the done call from the user task and give
      -- him the output parameters he wants
      ACCEPT done (new_context: OUT kernel.xtnded_uid;
                        status: OUT ret_status) DO
      END done;   -- Release the user from the rendevouz

      -- Notify the controller that we are finished so he
      -- can put us back on his free list of available tasks
      controller.op_done(CREATE, my_num);

   END LOOP;

END create_cntxt;    -- End create operation task body



--------------------- DELETE_CNTXT TASK BODY -----------------
TASK BODY delete_cntxt IS
   my_num: op_task_num;     -- Our task # in the controller
   status: ret_stat;        -- Operation status
   tally: NATURAL;          -- Vote accumulator
   host: kernel.xtnded_uid;-- Storage for host ids
   loc_copy: context;       -- Local storage for our context
   ret_header: rep_data;    -- Headers returned from remotes
   ---------------
   -- Define a local function to delete a context
   -- on the local host
   FUNCTION local_delete RETURN ret_stat IS
   BEGIN
      -- Copy the local context's header into ret_header
      -- Call the kernel to return the object's storage
      -- Return status of ERROR if the kernel balks
```

```
      -- Else return OK
   END local_delete;
   ---------------
   -- Define a local function to send a delete request to
   -- a remote host and get his response
   FUNCTION remote_delete(context_id: IN kernel.xtnded_uid;
                                orig: IN BOOLEAN;
                          which_host: IN kernel.xtnded_uid) IS
   BEGIN
      -- Call the support package to pack the context_id and
      --   other parameters
      -- Call the kernel to send the request to the host
      --   identified in the which_host parameter
      -- Wait for the response from the remote host
      -- Unpack the return data and put into ret_header
      -- Return the status received from the remote call
   END remote_delete;


BEGIN    -- Delete operation task body
   -- Accept the initialization call from the controller
   -- and remember our task number in my_num
   ACCEPT init (my_num: IN op_task_num) DO
   END init;

   LOOP     -- One operation at a time forever
      -- Accept the startup call from the controller and get
      -- the input parameters for the call
      ACCEPT start (context_id: IN kernel.xtnded_uid;
                          orig: IN BOOLEAN) DO
      END start;

      -- If this is the original call, we must act as
      -- collector for the replication algorithm
      IF orig THEN

            -- Start a transaction so we can be sure that a
            -- write quorum of the object copies get deleted
            TRANMGR.begin_transaction;

            -- Get the local copy into our memory space
            support.get_object(context_id, loc_copy);

            -- Perform the operation on each copy and tally the
            -- votes returned in the headers
            tally := 0;
            LOOP
               -- Get the next host from the list in the
               -- local copy's header. Exit when no more hosts
               host := support.next_host(loc_copy.header);
               EXIT WHEN host = NULL_UID;
```

```
            -- Perform a local or remote delete depending
            -- upon the next host id
            IF host = my_host THEN
               status := local_delete;
            ELSE
               status :=
                  remote_delete(context_id, FALSE, host);
            END IF;

            -- If the operation was successful, the votes
            -- for this copy are counted ONLY if it has the
            -- the most recent version number
            IF status = OK THEN
               IF ret_header.v = loc_copy.header.v THEN
                  tally := tally + ret_header.n;
               ELSEIF ret_header.v > loc_copy.header.v THEN
                  loc_copy.header.v := ret_header.v;
                  tally := ret_header.n;
               END IF;
            END IF;

         END LOOP;

         -- If the total number of votes accumulated from
         -- up-to-date copies is not at least equal to the
         -- write quorum for the object, abort the xaction
         IF tally < loc_copy.header.w THEN
            TRANMGR.abort_transaction;
            status := ERROR;
         ELSE
            status := OK;
         END IF;

         -- End the transaction and either abort or commit
         TRANMGR.end_transaction;

      END IF;

   -- If this is a subordinate call, just do it
   ELSE
      status := local_delete;

   END IF;


   -- Accept the done call from the user task and give
   -- him the output parameters he wants
   ACCEPT done (status: OUT ret_status) DO
   END done;

   -- Notify the controller that we are finished
   controller.op_done(DELETE, my_num);
```

```
   END LOOP;

END delete_cntxt;     -- End delete operation task body



-------------------- ADD_NAME TASK BODY --------------------
TASK BODY add_name IS
   my_num: op_task_num;      -- Our task # in the controller
   status: ret_stat;         -- Operation status
   tally: NATURAL;           -- Vote accumulator
   host: kernel.xtnded_uid;-- Storage for host ids
   loc_copy: context;        -- Local storage for our context
   ret_header: rep_data;     -- Headers returned from remotes
   ---------------
   -- Define a local function to add a name/uid pair to
   -- a context on the local host
   FUNCTION local_add RETURN ret_stat IS
   BEGIN
      -- Copy the local context's header into ret_header
      -- Call the hash package to add the name/uid pair
      --   to the context
      -- Return status of ERROR if the name is already in the
      --   context or if the hash package returns error
      -- Else return OK
   END local_add;
   ---------------
   -- Define a local function to send a add request to
   -- a remote host and get his response
   FUNCTION remote_add(context_id: IN kernel.xtnded_uid;
                             name: IN symb_name;
                          name_id: IN kernel.xtnded_uid;
                             orig: IN BOOLEAN;
                       which_host: IN kernel.xtnded_uid) IS
   BEGIN
      -- Call the support package to pack the context_id and
      --   other parameters
      -- Call the kernel to send the request to the host
      --   identified in the which_host parameter
      -- Wait for the response from the remote host
      -- Unpack the return data and put into ret_header
      -- Return the status received from the remote call
   END remote_add;


BEGIN    -- Add operation task body
   -- Accept the initialization call from the controller
   -- and remember our task number in my_num
   ACCEPT init (my_num: IN op_task_num) DO
   END init;
```

```
LOOP     -- One operation at a time forever
   -- Accept the startup call from the controller and get
   -- the input parameters for the call
   ACCEPT start (context_id: IN kernel.xtnded_uid;
                       name: IN symb_name;
                    name_id: IN kernel.xtnded_uid;
                       orig: IN BOOLEAN) DO
   END start;

   -- If this is the original call, we must act as
   -- collector for the replication algorithm
   IF orig THEN

         -- Start a transaction so we can be sure that a
         -- write quorum of the object copies get updated
         TRANMGR.begin_transaction;

         -- Get the local copy into our memory space
         support.get_object(context_id, loc_copy);

         -- Perform the operation on each copy and tally the
         -- votes returned in the headers
         tally := 0;
         LOOP
            -- Get the next host from the list in the
            -- local copy's header. Exit when no more hosts
            host := support.next_host(loc_copy.header);
            EXIT WHEN host = NULL_UID;

            -- Perform a local or remote add depending
            -- upon the next host id
            IF host = my_host THEN
               status := local_add;
            ELSE
               status :=
                  remote_add(context_id,
                                   name,
                                name_id,
                                  FALSE,
                                   host);
            END IF;

            -- If the operation was successful, the votes
            -- for this copy are counted ONLY if it has the
            -- the most recent version number
            IF status = OK THEN
               IF ret_header.v = loc_copy.header.v THEN
                  tally := tally + ret_header.n;
               ELSEIF ret_header.v > loc_copy.header.v THEN
                  loc_copy.header.v := ret_header.v;
                  tally := ret_header.n;
               END IF;
            END IF;
```

```
            END LOOP;

            -- If the total number of votes accumulated from
            -- up-to-date copies is not at least equal to the
            -- write quorum for the object, abort the xaction
            IF tally < loc_copy.header.w THEN
               TRANMGR.abort_transaction;
               status := ERROR;
            ELSE
               status := OK;
            END IF;

            -- End the transaction and either abort or commit
            TRANMGR.end_transaction;

         END IF;

      -- If this is a subordinate call, just do it
      ELSE
         status := local_add;

      END IF;

      -- Accept the done call from the user task and give
      -- him the output parameters he wants
      ACCEPT done (status: OUT ret_status) DO
      END done;

      -- Notify the controller that we are finished
      controller.op_done(ADD, my_num);

   END LOOP;

END add_name;      -- End add operation task body




--------------------- REMOVE_NAME TASK BODY ------------------
TASK BODY remove_name IS
   my_num: op_task_num;      -- Our task # in the controller
   status: ret_stat;         -- Operation status
   tally: NATURAL;           -- Vote accumulator
   host: kernel.xtnded_uid;-- Storage for host ids
   loc_copy: context;        -- Local storage for our context
   ret_header: rep_data;     -- Headers returned from remotes
   ---------------
   -- Define a local function to remove a name/uid pair
   -- from  a context on the local host
   FUNCTION local_remove RETURN ret_stat IS
```

```
   BEGIN
      -- Copy the local context's header into ret_header
      -- Call the hash package to remove the pair from
      --   the context
      -- Return status of ERROR if the name is not found
      -- Else return OK
   END local_remove;
   ---------------
   -- Define a local function to send a remove request to
   -- a remote host and get his response
   FUNCTION remote_delete(context_id: IN kernel.xtnded_uid;
                                name: IN symb_name;
                                orig: IN BOOLEAN;
                          which_host: IN kernel.xtnded_uid) IS
   BEGIN
      -- Call the support package to pack the context_id and
      --   other parameters
      -- Call the kernel to send the request to the host
      --   identified in the which_host parameter
      -- Wait for the response from the remote host
      -- Unpack the return data and put into ret_header
      -- Return the status received from the remote call
   END remote_remove;


BEGIN    -- Remove operation task body
   -- Accept the initialization call from the controller
   -- and remember our task number in my_num
   ACCEPT init (my_num: IN op_task_num) DO
   END init;

   LOOP     -- One operation at a time forever
      -- Accept the startup call from the controller and get
      -- the input parameters for the call
      ACCEPT start (context_id: IN kernel.xtnded_uid;
                          name: IN symb_name;
                          orig: IN BOOLEAN) DO
      END start;

      -- If this is the original call, we must act as
      -- collector for the replication algorithm
      IF orig THEN

            -- Start a transaction so we can be sure that a
            -- write quorum of the object copies get updated
            TRANMGR.begin_transaction;

            -- Get the local copy into our memory space
            support.get_object(context_id, loc_copy);

            -- Perform the operation on each copy and tally the
            -- votes returned in the headers
            tally := 0;
```

```
      LOOP
         -- Get the next host from the list in the
         -- local copy's header. Exit when no more hosts
         host := support.next_host(loc_copy.header);
         EXIT WHEN host = NULL_UID;

         -- Perform a local or remote remove depending
         -- upon the next host id
         IF host = my_host THEN
            status := local_remove;
         ELSE
            status :=
               remote_remove(context_id,
                                    name,
                                   FALSE,
                                    host);
         END IF;

         -- If the operation was successful, the votes
         -- for this copy are counted ONLY if it has the
         -- the most recent version number
         IF status = OK THEN
            IF ret_header.v = loc_copy.header.v THEN
               tally := tally + ret_header.n;
            ELSEIF ret_header.v > loc_copy.header.v THEN
               loc_copy.header.v := ret_header.v;
               tally := ret_header.n;
            END IF;
         END IF;

      END LOOP;

      -- If the total number of votes accumulated from
      -- up-to-date copies is not at least equal to the
      -- write quorum for the object, abort the xaction
      IF tally < loc_copy.header.w THEN
         TRANMGR.abort_transaction;
         status := ERROR;
      ELSE
         status := OK;
      END IF;

      -- End the transaction and either abort or commit
      TRANMGR.end_transaction;

   END IF;

-- If this is a subordinate call, just do it
ELSE
   status := local_remove;

END IF;
```

```
      -- Accept the done call from the user task and give
      -- him the output parameters he wants
      ACCEPT done (status: OUT ret_stat) DO
      END done;

      -- Notify the controller that we are finished
      controller.op_done(REMOVE, my_num);

   END LOOP;

END remove_name;      -- End remove operation task body




--------------------- LOOKUP_NAME TASK BODY ------------------
TASK BODY lookup_name IS
   my_num: op_task_num;      -- Our task # in the controller
   status: ret_stat;         -- Operation status
   tally: NATURAL;           -- Vote accumulator
   host: kernel.xtnded_uid;-- Storage for host ids
   loc_copy: context;        -- Local storage for our context
   ret_header: rep_data;     -- Headers returned from remotes
   name_uid: kernel.xtnded_uid;  -- UID associated with name
   ret_uid: kernel.xtnded_uid;   -- UID returned by remotes
   ---------------
   -- Define a local function to lookup a name/uid pair
   -- in a context on the local host
   FUNCTION local_lookup RETURN ret_stat IS
   BEGIN
      -- Copy the local context's header into ret_header
      -- Call the hash package to find the name in context
      -- Put the associated UID in ret_uid
      -- Return status of ERROR if name was not found
      -- Else return OK
   END local_lookup;
   ---------------
   -- Define a local function to send a lookup request to
   -- a remote host and get his response
   FUNCTION remote_lookup(context_id: IN kernel.xtnded_uid;
                                name: IN symb_name;
                                orig: IN BOOLEAN;
                          which_host: IN kernel.xtnded_uid) IS
   BEGIN
      -- Call the support package to pack the context_id and
      --   other parameters
      -- Call the kernel to send the request to the host
      --   identified in the which_host parameter
      -- Wait for the response from the remote host
      -- Unpack the return header and put into ret_header
      -- Put the returned UID in ret_uid
```

```
      -- Return the status received from the remote call
   END remote_lookup;


BEGIN    -- Lookup operation task body
   -- Accept the initialization call from the controller
   -- and remember our task number in my_num
   ACCEPT init (my_num: IN op_task_num) DO
   END init;

   LOOP     -- One operation at a time forever
      -- Accept the startup call from the controller and get
      -- the input parameters for the call
      ACCEPT start (context_id: IN kernel.xtnded_uid;
                          name: IN symb_name;
                          orig: IN BOOLEAN) DO
      END start;

      -- If this is the original call, we must act as
      -- collector for the replication algorithm
      IF orig THEN

            -- Start a transaction so we can be sure that a
            -- read quorum of the object copies get examined
            TRANMGR.begin_transaction;

            -- Get the local copy into our memory space
            support.get_object(context_id, loc_copy);

            -- Perform the operation on each copy and tally the
            -- votes returned in the headers
            tally := 0;
            LOOP
               -- Get the next host from the list in the
               -- local copy's header. Exit when no more hosts
               host := support.next_host(loc_copy.header);
               EXIT WHEN host = NULL_UID;

               -- Perform a local or remote lookup depending
               -- upon the next host id
               IF host = my_host THEN
                  status := local_lookup;
               ELSE
                  status :=
                     remote_lookup(context_id,
                                   name,
                                  FALSE,
                                   host);
               END IF;

               -- If the operation was successful, the votes
               -- for this copy are counted
```

```
                -- The returned UID of the copy with the highest
                --   version number is retained
                IF status = OK THEN
                   tally := tally + ret_header.n;
                   IF ret_header.v >= loc_copy.header.v THEN
                      loc_copy.header.v := ret_header.v;
                      name_uid := ret_uid;
                   END IF;
                END IF;

             END LOOP;

             -- If the total number of votes accumulated from
             -- up-to-date copies is not at least equal to the
             -- read quorum for the object, abort the xaction
             IF tally < loc_copy.header.r THEN
                TRANMGR.abort_transaction;
                status := ERROR;
             ELSE
                status := OK;
             END IF;

             -- End the transaction and either abort or commit
             TRANMGR.end_transaction;

          END IF;

       -- If this is a subordinate call, just do it
       ELSE
          status := local_lookup;

       END IF;

       -- Accept the done call from the user task and give
       -- him the output parameters he wants
       ACCEPT done (name_uid: OUT kernel.xtnded_uid;
                    status: OUT ret_stat) DO
       END done;

       -- Notify the controller that we are finished
       controller.op_done(LOOKUP, my_num);

    END LOOP;

 END lookup_name;      -- End lookup operation task body




 -------------------- CALL HANDLER BODY ---------------------
 TASK BODY rem_call IS
```

```
    TASK TYPE wait_create IS    -- Example wait task definition
       ENTRY init (my_num);
       ENTRY start (...);
    END wait_task;

    TASK BODY wait_create IS
    BEGIN
       ACCEPT init(my_num) DO  -- Remember our task number
       END init;

       LOOP
          ACCEPT start (...) DO   -- Fire up and get parameters
          END start;

          -- Call controller to do the operation
          controller.create_cntxt(..., create_task);

          -- Wait for the create task to complete
          create_task.done(...);

          -- Call the kernel to send the response over the RPC

          -- Call remote call handler to tell him we're done

       END LOOP;

    END wait_create;

    -- Define arrays of wait tasks

 BEGIN    -- Remote call handler body

    LOOP
       -- Call the kernel to get a remote call

       -- Schedule a wait task and call its start entry
             passing the parameters from the RPC

       -- Return the wait task to the queue when its finished

    END LOOP;

 END rem_call;  -- End of remote call handler body




END SNTM;     -- End of type manager package body
```

Chapter 6

MESSAGE TYPE MANAGER DESIGN

6.1 INTRODUCTION

The Message Type Manager (MTM) is replicated on hosts in the network wherever inter-process communication by messages is desired. The instances of the MTM are identical. The composition of a MTM and its interface to the user is shown in Figure 6-1. User operation requests are made to the controller of the MTM which takes the appropriate action. A task to perform the requested operation is scheduled by the MTM_Controller from a pool of SEND, RECEIVE and MSG_STATUS tasks. (In the subsequent text, all capitalized words will refer to tasks of the MTM).

In sending a message, SEND calls the creation operation of MESSAGE OBJECT, which returns a message object, routes copies of the message object to remote hosts (as determined by the reliability class), and sends notices of availability to the intended receivers. A notice becomes an entry in the message queue for a receiver process. If the call is asynchronous, the message identifier is then returned to the sender. If the call is synchronous, SEND terminates but the sender remains blocked until SUPPORTER determines that some event has occurred and causes the sender to proceed (i.e. after a timeout or after all acknowledgements of copies sent are returned). When a sender is to be unblocked, the SUPPORTER schedules a WAKER task to bundle and route the appropriate response to the sender.

RECEIVE determines from PROCESS MESSAGE QUEUE (PMQ) whether or not a message is available that meets the specifications of the receiver. The PMQ manages all message queues for the processes of that host. It maintains the queues in stable storage. If there is a message available, it is returned to the receiver and RECEIVE terminates. If there is no message available and the call is asynchronous RECEIVE terminates and the receive continues without having received a message. In a synchronous call the sender remains blocked while SEND terminates and SUPPORTER performs the detection of the event to resume the receiver (either a timeout or an appropriate message arriving for the receiver). A WAKER task bundles a response and routes it to a waiting receiver.

The MSG_STATUS task returns the status of a message that is retrieved from a local copy of the message (if there is one), otherwise the status is returned from a remote copy.

One facet of the MTM not depicted in Figure 6-1 concerns the routine acknowledgement of events between MTMs regarding the routing of message copies. This is MTM_Controller to MTM_Controller communication via Kernel remote calls and responses. Some of the communications cause interactions with SUPPORTER (i.e. such an acknowledgement for a message copy sent to a

remote host). Another interaction occurs when the PMQ is notified by its MTM_Controller of incoming notices of message availability for receivers on that host. When a copy of a message is required on a host where no such copy exists a request is made to a remote host that has a copy which causes a message copy to be routed to the requesting host. Such interactions occur between MTM_Controllers and are necessary for the smooth functioning of the operations of the MTM.

6.2 CONSISTENCY AND RELIABILITY MECHANISMS FOR MESSAGES

Messages provide reliable inter-process communication in either a synchronous or an asynchronous fashion within the ZEUS Operating System. To increase both the reliability and the accessibility of a message object, the message object may be a multiple copy object with copies existing on different hosts (preferably on hosts where it will be received). Since the message is basically an immutable object, there is no "primary" copy - any copy of the object may be received by an authorized process (one on the message receive list). A reliable underlying communication system is assumed.

It is desirable that the state of a message object remain consistent for as much of the time as is possible. This is because the state of the entire network is reflected in the messages that exist in the system at any given moment. The consistency of a message object is reflected in how it appears to the intended receivers of the message. At the highest degree of consistency, the state of the object appears the same at any instant. The state of the object (i.e., each of its copies) is either available for receipt to all intended receivers or not available for receipt. It is impossible to achieve this highest level of consistency since the simultaneous message creation and availability to all intended receivers in the network is not possible. The next best thing is to make as small as possible the period in which the state of the message object is inconsistent, and to have the means to render a message object consistent from an inconsistent state due to failure.

Message object consistency is maintained in an error free environment by protocols or conventions that are obeyed by each MTM with regard to the maintenance of multiple copy message objects. These conventions will be discussed in the context of the message operations. In an environment in which a failure has occurred, such as the crash of a host, efforts on behalf of alive hosts can render an inconsistent message object consistent.

This chapter discusses (1) how message object consistency is ensured by the message operations of an MTM in an error free environment and (2) how consistency is maintained in the event of host failure(s).

Looking at each operation in turn, the send operation invoked by a process causes a message object to be created and one or more copies of the object are distributed on hosts in the network. A copy consists of a count of the number of copies and the receiver list for the message. Obviously, the more copies that are made of a message, the more reliable the message since more copies increase the likelihood that it will be available if a failure occurs. The least reliable is a one copy message since a host failure of the copy host results in the unavailability of the message until the host rejoins

the network (which is not a problem if all intended receivers also exist on that host).

The message copies may be maintained on either volatile or non-volatile storage, the determination of which is made depending on the reliability class of the message object. The four reliability classes for an object as defined in Chapter 2 of Volume 1 of the guidebook are (1) volatile, (2) non-volatile, (3) resilient and (4) stable. Resilient objects require recovery mechanisms and are described below. Message creation can include "object reliability class" as a parameter. The characteristics of the message objects relative to the four classes is discussed.

Volatile objects are guaranteed to be consistent only in the absence of failures. In the context of message objects, this is analogous to single-copied message objects that are stored on volatile storage. The failure of the host will render the object inconsistent - in fact, it will no longer exist. These objects have the "non-guaranteed delivery" property.

A non-volatile object is one that has multiple volatile copies or a single non-volatile copy of the message object. If the object was consistent before a failure then it is guaranteed to be consistent after that failure. No recovery actions are taken for non-volatile objects.

Resilient objects have recovery operations performed to change any object rendered inconsistent due to failure to a consistent state after recovery. The object is returned to the most previous consistent state which for messages means the message object will be backed out when the object state is inconsistent due to a failure during the send operation. In this case, the send operation is considered to be incomplete. The send operation is the only inconsistent state producing message operation.

Stable message objects continue to be "accessible and consistent even while the failure is being repaired." This is supported by requiring that one copy of the message be put on every host so that a copy of the message is always available even in the event of failures that result in the availability of only one host in the network. Consistency of stable objects is maintained in the same fashion as for resilient objects.

The send operation causes a message object to be created and one or more copies of the object to be distributed across the network. The number of copies to be distributed is determined by a parameter at the message interface level between the calling process and the MTM. The message queue of each intended receiver is updated to include the message id of the available message and the message copies are distributed to various hosts depending on the reliability class of the message object.

It is possible for the state of a message object to be rendered inconsistent when the send operation is not completed due to failure. In this case, the send operation is incomplete. The inconsistent state might be that not all copies of the message are distributed, or that only some but not all of the process message queues have been updated to reflect the availability of

the message. Some recovery action must be taken by the alive hosts to make such a message state is consistent, or to back out the message.

Given the types of inconsistencies that can occur due to the interruption of a send operation, the failure and recovery process will be viewed in light of (1) error detection, (2) damage retention, (3) error recovery, and (4) error correction. In its simplest form, error detection occurs when some kernel detects a host failure and broadcasts this information to the network.

When a failure occurs, these are the possible states characterizing a message being sent:

o The message queues of the receiver processes reflect the message id and all copies of the message have been distributed.

  This is a consistent state. No recovery action is necessary.

o Not all message queues of the receiver processes reflect the message id of the last message sent: This is an inconsistent state and the message will be backed out, i.e. the message queues with the message id will have that message id removed.

o The message id of the last message sent is in the message queue of each available process, and one or more copies of the message are distributed.

  Any remaining copies to be distributed will not be distributed. But as long as one copy is available it can be received by any receiver.

o If there is no copy available because the only copy is on the downed host, then the message is unavailable to any receivers on the remaining hosts.

  No recovery action can be taken and the message cannot be received because no copy is available. It cannot be backed out because it is possible that the message has been received on the downed host.

Each message operation is performed as a transaction. Because of this, the operation either executes to completion or is backed out by the Transaction Manager so that no inconsistent state is produced.

The receive operation is most sensitive at the point of receipt since it is desirable that te recipt of he message be made apparent on the message copy which is received at the moment it is received. When the receive process has received the message and returned it to the process invoking the receive, the message copy that was received is updated to reflect the receipt of the message.

The message status operation is a read only operation and does not require any type of recovery operation since it never causes an inconsistency in a message object.

6.3 HIGH-LEVEL DESCRIPTION OF MTM MODULES

The Zeus Operating System is being designed with the purpose of using and analyzing reliability and recovery mechanisms for distributed systems as the basis for the preparation of a guidebook oriented towards the designer's of reliable distributed systems. As an object-oriented system, Zeus allows the specification of reliability on an object by object basis, so that some objects may have a high reliability and others a lower reliability by option. The Message Type Manager (MTM) manages objects of type message and is responsible for performing the operations send, receive, and message_status for a message object. Messages are a means of communicating between processes. For a particular message object, the MTM ensures that its reliability class is maintained according to the specifications of its creator.

The MTM is a replicated object because there are multiple MTMs existing in the network with at most one MTM on any given host. There may be fewer MTMs than there are hosts in the sys- tem, in which case it is proposed that a "stub" MTM exists on any host that does not have a MTM. A stub MTM would route the message operation invocations from the processes on that host to other MTMs to perform the operations, and to transfer the results of the operation to the process that invoked the operation. Currently, only a MTM is designed. A stub MTM would be a subset of the MTM. The complete MTM is discussed in this chapter.

As seen by a user process, the operations on a message object are send, receive, and get the message status. The send operation causes a message to be created and sent to another process. Like other objects in the Zeus system, a message object is given a unique identifier when it is created. The receive operation is the means by which a process reads a message which it has been sent. It can only read a message once. The msg_status operation returns the status of a particular message, which may be received, not received, unavailable, or non- existent.

The components of message management are depicted in Figure 6-1. The user process shown at top contains a MTM interface for operation invocation and for the receipt of the results. The large box below the user encloses the components of the Message Type Manager. Invocations are made from the MTM_Interface to the MTM_Controller which hands off the request to one of the send, receive, or the msg_status processes. When the operation is being performed, the services of the Process_Message_Queue and the Message_Object, respectively, provide information about the messages outstanding for a process and also the location and characteristics of a message object. When the operation is completed a result is returned to the MTM_Interface.

Functionally, the synchronous and asynchronous send and receive message operations are quite similar. A synchronous send means "send and wait until received" and a receive means "wait until there is a message to be received." An asynchronous call has neither of these waits. The Supporter and Waker processes are key in the achievement of synchronicity. The Supporter is notified of events and determines when synchronization has occurred or, alternatively, a timeout. A Waker task returns a result to a waiting process.

The three directed arrows in Figure 6-1 between the MTM_Interface and the Message Type Manager are expanded in Figure 6-2 into a series of routines and pathways that express at a greater level of detail the interface between the user and the MTM. In this chapter, each of these components is described beginning with the user interface routines and the task of the MTM_Interface shown in Figure 6-2 and followed by the components of the Message Type Manager shown in Figure 6-1.

When a user invokes a message operation, a procedure call is made to one of the send, receive, or msg_status procedures within the MTM_Interface. Figure 6-2 shows these routines. From these procedures an entry call [labeled (1)] is made to the Message_Operations task which is responsible for the protocol between the MTM_Interface and the MTM. From this task, an entry call at (2) is made to the MTM_Controller that causes one of the send, receive, or msg_status processes of Figure 6-1 to become active. When the operation completes, a result is sent from one of the send, receive, msg_status, or waker processes to the Message_Operations task where its receipt causes the user to become unblocked with the results of the operation.

The following sections numbered 6.3.1 through 6.3.4 are (respectively) the descriptions of the MTM_Interface components Send_Msg, Receive_Msg, Msg_Status, and Message_Operations.

6.3.1 Send_Msg Procedure

The user interface to perform message manipulations is a procedural one. Calls are made to routines named send_msg, receive_msg and msg_status that are part of a Message_Interface package within the process space of the user. The following are the procedural interfaces between the user process and the Message_Interface routines.

```
PROCEDURE send_msg (msg_vars:  IN MTM_type.parm_list;
                send_to_list:  IN MTM_type.xid_list;
                      option:  IN MTM_type.wait_no_wait;
                     timeout:  IN POSITIVE;
         reliability_class:  IN MTM_type.rel_classes;
                      msg_id:  OUT kernel.xtnded_uid;
               return_status:  OUT MTM_type.msg_opn_return);
```

Parm_list is a record that describes the variables that compose a message. Some convention will be made between the compiler(s) of a host machine and the send_msg procedure as to the actual record description of parm_list.

The send_to_list is a linked list of the intended receivers of the message. A broadcast of a message is indicated when the send_to_list is composed of a single star, "*".

The process has the option of waiting for acknowledgements that the message has been sent to every receiver or not waiting for the acknowledgements. This is specified by "wait" or "no_wait" as the value of the option.

If the option is wait, a timeout value must be specified which is the maximum time that the sender is willing to wait for the acknowledgements.

The reliability_class for a message object may be volatile, non-volatile, resilient, or stable. A volatile message object is one with the least likelihood of being available if some failure occurs because it is a single copy object in memory. A stable message object has the greatest likelihood of being available because a copy of the message exists on each host, and is thus a replicated object. Non_volatile and resilient message objects are more reliable than volatile objects and less reliable than stable objects. The number of message copies created during a send operation and their storage medium will be varied during performance analysis to determine what combinations provide the maximum amount of reliability and efficiency. One major difference between non_volatile and resilient is that non_volatile objects have no recovery operations performed for them upon failure, but resilient objects do.

The msg_id is a unique identifier for the message that is returned after the message is sent. This identifier may be used in a msg_status call to determine the state of the message regarding its receipt.

The return_status contains the result of the send operation and may be completed or timed_out. The not_completed status will at a later time be expanded into a group of possible error return values according to the fault that caused the operation to fail.

6.3.2 Receive_Msg Procedure

The complement to send_msg operation is the receive_msg operation that a process invoked to receive a message that is available.

```
PROCEDURE receive_msg (msg_vars:  IN MTM_type.parm_list;
                 receive_from_list:  IN MTM_type.xid_list;
                       wait_option:  IN MTM_type.wait_no_wait;
                  which_msg_option:  IN receive_option;
                           timeout:  IN POSITIVE;
                            msg_id:  OUT kernel.xtnded_uid;
                         sender_id:  OUT kernel.xtnded_uid;
                     return_status:  OUT MTM_type.msg_opn_return);
```

Msg_vars are the variables into which a received message is placed.

The receive_from_list indicates which process the receiver is willing to receive from. It may be a linked list of process extended uids, or a star (*) which indicates a willingness to receive from any process.

The wait_option may have the values of either wait or no_wait where wait will cause the receiver to wait a finite amount of time for a message to arrive, the wait time being indicated by timeout.

The which_msg_option may be either most_recent, oldest, or first_after_failure. This gives the receiver flexibility in receiving messages.

The msg_id contains the extended uid of the just received message.

The sender_id contains the extended uid of the process that sent the message.

The return_status may be completed or timed_out.

6.3.3 Msg_Status Procedure

The current status of any particular send message operation may be determined with the msg_status operation.

```
  PROCEDURE msg_status (msg_id:  IN kernel.xtnded_uid;
              return_statuses:  OUT MTM_type.msg_opn_return_list);
```

The msg_id is the extended uid of the message for which a status query is being made.

The return_status record is a linked list of process_id/status pairs. That is, one status is returned for each intended receiver process. The possible return statuses are received, not_received, unavailable (i.e., status not known), and non_existent.

This completes the discussion of the procedural interface to the message operations.

6.3.4 Message_Operations Task

The three procedures (send_msg, receive_msg, and msg_status) make entry calls: to a Message_Operations task that is the Message_Interface of the calling process to the Message Type Manager of that host. There is one Message_Operations task for each user process that is part of the Message_Interface.

The Message_Operations task is very simple and accepts one of the entry calls route_send_msg, route_receive_msg, or route_msg_status. Once the entry_call is accepted, the Message Operations task makes an entry call to the MTM_controller that effectively requests that the message operation be performed. After making the request, the Message_Operations task waits to accept an entry call containing the result.

```
TASK message_operations IS

  ENTRY route_send_msg (text:  IN kernel.message;
                process_list:  IN MTM_type.xid_list;
           reliability_class:  IN MTM_type.rel_classes;
                      option:  IN MTM_type.wait_no_wait;
                     timeout:  IN POSITIVE;
```

```
                 msg_id:  OUT kernel.xtnded_uid;
          return_status:  OUT MTM_type.msg_opn_return);

  ENTRY send_return (msg_id:  IN kernel.xtnded_uid;
              return_status:  IN MTM_type.msg_opn_return);

  ENTRY route_receive_msg (process_list:  IN MTM_type.xid_list;
                            wait_option:  IN MTM_type.wait_no_wait;
                       which_msg_option:  IN MTM_type.receive_option;
                                timeout:  IN POSITIVE;
                                 msg_id:  OUT kernel.xtnded_uid;
                              sender_id:  OUT kernel.xtnded_uid;
                                   text:  OUT kernel.message;
                          return_status:  OUT MTM_type.msg_opn_return);

  ENTRY receive_return (msg_id:  IN kernel.xtnded_uid;
                     sender_id:  IN kernel.xtnded_uid;
                          text:  IN kernel.message;
                 return_status:  IN MTM_type.msg_opn_return);

  ENTRY route_msg_status (msg_id:  IN kernel.xtnded_uid
                   return_status:  OUT MTM_type.msg_opn_return_list);

  ENTRY msg_status_return (return_statuses:  IN MTM_type.msg_opn_return_
                           list);

END message_operations;
```

One might ask whether or not it would be simpler and/or more efficient to make the entry call to the MTM_Controller directly from the Message_Interface procedures and avoid having the Message_Operations task altogether. However, a call to perform a message operation is a synchronous call such that the calling process is blocked until some result is returned (the semantics are the same as for a procedural call). So if the calling process makes an entry call into the MTM_Controller, the MTM_Controller would effectively have to wait for the operation to be completed to keep the calling procedure blocked and to return a result via the initial entry call. This effectively causes all message operations for that host to be performed in a sequential fashion which is not a viable alternative.

Additionally, and perhaps most importantly, the result to an operation may be returned from either one of the operation tasks (send, receive, or msg_status) or the waker task (which may return the result when an operation is performed with the wait option). A diagram containing these tasks and the entry calls they make in Message_Operations is given in Figure 6-3. When the initial operation invocation is made, it is not known from what task the result will come, so the interface task performs an ACCEPT for the result which allows the result to come from any task. An ACCEPT can only be performed from a task which is a main motivation for making Message_Operations a task, rather than limiting the interface to the three previously described proce- dures.

6.3.5 MTM_Controller Task

The MTM_Controller accepts requests for the message opera- tions to be performed and also handles MTM intercommunication. The body of the MTM_Controller is a loop in which it accepts entry calls to perform some action. Figure 6-4 groups the entries by class and associates with each entry the task scheduled by a rendezvous at that entry. The Message Operations Class are those entries associated with the invocation of opera- tions. The Acks Class of entries are acknowledgements of opera- tion completion. Routing Copies are used in message object movement between hosts. Finally, the Message Queue Class of entries is related to the management for processes of the message queue of message object identifiers.

```
TASK MTM_controller IS

  ENTRY failure_notice ( );

  ENTRY send_msg (sender:  IN kernel.xtnded_uid;
                    text:  IN kernel.message;
            process_list:  IN MTM_type.xid_list;
       reliability_class:  IN MTM_type.rel_classes;
                  option:  IN MTM_type.wait_no_wait;
        which_msg_option:  IN MTM_type.receive_option;
                  timeout:  IN POSITIVE;
                call_uid:  IN kernel.uid);

  ENTRY receive_msg (receiver:  IN kernel.xtnded_uid;
                 process_list:  IN MTM_type.xid_list;
                         text:  IN kernel.message;
                  wait_option:  IN MTM_type.wait_no_wait;
             which_msg_option:  IN MTM_type.receive_option;
                      timeout:  IN POSITIVE;
                     call_uid:  IN kernel.uid);

  ENTRY msg_status (requestor:  IN kernel.xtnded_uid;
                       msg_id:  IN kernel.xtnded_uid;
                     call_uid:  IN kernel.uid);

  ENTRY get_send_parms (sender:  IN kernel.xtnded_uid;
                          text:  IN kernel.message;
                   sender_list:  IN MTM_type.xid_list;
             reliability_class:  IN MTM_type.rel_classes;
                        option:  IN MTM_type.wait_no_wait;
                       timeout:  IN POSITIVE;
                      call_uid:  IN kernel.uid);

  ENTRY get_receive_parms (receiver:  IN kernel.xtnded_uid;
                               text:  IN kernel.message;
                       receive_list:  IN MTM_type.xid_list;
                        wait_option:  IN MTM_type.wait_no_wait;
                   which_msg_option:  IN receive_option;
                            timeout:  OUT POSITIVE;
                           call_uid:  IN kernel.uid);
```

```
ENTRY get_msg_status_parms (requester:  IN kernel.xtnded_uid;
                               msg_id:  IN kernel.xtnded_uid;
                             call_uid:  IN kernel.uid);

ENTRY message_copy_for_receiver (msg_id:  IN kernel.xtnded_uid;
                             msg_object:  IN MTM_type.object);

ENTRY msg_received (msg_id:  IN kernel.xtnded_uid;
                  receiver:  IN kernel.xtnded_uid);

ENTRY request_copy (msg_id:  IN kernel.xtnded_uid;
          MTM_requestor_id:  IN kernel.xtnded_uid;
                  call_uid:  IN kernel.uid);

ENTRY send_copy (msg_object:  IN MTM_type.object;
             storage_medium:  IN MTM_type.storage_type;
                   call_uid:  IN kernel.uid);

ENTRY send_copy_ack (msg_id:  IN kernel.xtnded_uid;
                     sender:  IN kernel.xtnded_uid);

ENTRY send_notice (msg_id:  IN kernel.xtnded_uid;
             notice_group:  IN MTM_type.xid_list;
                 call_uid:  IN kernel.uid); END MTM_controller;
```

The relationship between entries of the MTM_Controller and tasks which call those entries is given in Figure 6-5.

The failure_notice is the means by which the MTM_Controller is informed of restart after a failure. Because the operations of the MTM are performed as transactions, the recovery actions required for operations is managed by the Transaction Manager. The process message queues of the MTM must be "recovered" however, so the failure_notice causes the process_message_queue task to be appropriately notified.

Operations are invoked by entry calls from Message_Operations (the MTM interface task to the user) to the MTM_Controller in the form of send_msg, receive_msg, and msg_status. The MTM_Controller schedules the appropriate process to perform the operation and passes the parameters to the process by get_send_parms, get_receive_parms, or get_msg_status_parms, respectively.

The message_copy_for_receiver is the entry by which a message instance is passed to this host as the result of a request from this host for the message instance. This is an instance that is sent "on demand", and occurs only when a message available for receipt is not on the same host as one of its intended receivers.

The msg_received entry is an acknowledgement that a particular message instance was received by a receiver on a remote host.

When a message instance 's "demanded", the request_copy entry is made which causes a message instance to be routed to the "message_copy_for_receiver" entry of the demanding host.

A send process may route instances of a message object for storage on other hosts in the network. A copy is sent to the send_copy entry of a remote host.

The send_copy_ack is the acknowledgement that a message instance sent by a SEND process has been stored on the remote host.

One or more process message queues on a host are updated as a result of send_notices from a send process.

The code is designed with efficiency in mind so that message operations may be scheduled and executed in parallel.

The following sections, 6.3.6 through 6.3.13, are descrip- tions of the tasks that are the component tasks of Figure 6-1. These tasks are also listed in the table of Figure 6-6. The tasks Remote_Receive_Call and Remote_Receive_Response listed in

Figure 6-6 are the interface tasks to the kernel network interface.

6.3.6 Send Task

A send task begins executing by getting the parameters from the MTM_Controller for that particular instance of the operation. A kernel call is made to get a unique identifier for the message that is to be created. Given the reliability class of the message being created, and also the send_to_list, a determination is made of the hosts to be sent a copy of the message, as well as the storage medium for the message. During performance analysis, the determination of copy_hosts will vary in order to study the effects of message copy distribution and availability.

The message object is created, and a notice of message availability is sent to each MTM, so that the appropriate process message queues may be updated to reflect the message availability. Following this is the sending of the message copy to the "copy_hosts" previously mentioned. At this point, if the option on the call is no_wait, a return status is routed to the sender by an entry call into the Message_Operations task of the sender.

If the option is wait, then an entry call is made to the supporter task that requests the sender be awakened when either the timeout on the call occurs, or each message copy sent to other hosts have arrived at their targeted hosts. Acknowledgements are sent to the supporter task from the MTM_Controllers that receive a message copy.

6.3.7 Receive Task

Like the send task, a receive task begins execution by getting the parameters for that receive operation from the MTM_Controller. The receiver's message queue is checked to determine if a message is available for that

process. If a message is not available, and the option on the call specifies that a wait should be made, then the supporter task is notified that the receiver process is waiting for a message. The supporter task will awaken the receiver when either the timeout occurs or a message is received (whichever occurs first). If the option is no_wait, and there is no message, then that status is returned to the receiver.

When a message is available and a copy of the message is on the same host as the receiver, it is routed to the receiver through an entry call to the Message_Operations task of the receiver process. The term "directed receive" is used to describe the receipt of a message object that is on the same host as the receiver. A message object may not be on the same host as each intended receiver since the reliability class of a message object might not require that each receiving host have a copy of the message. Figure 6-7 shows the entry calls made to achieve a directed receive.

If the message is not available on that host, the host hint in the message id from the receiver's available message id queue is used to determine where to get a copy of the message. The action of getting a copy of a message object from another host is termed "receive on demand" in this chapter, and the sequence of actions between the sender, the receiver, and their respective MTM_Controllers is shown in Figure 6-8. If that host is down, or for some reason that host's message copy is destroyed, a broadcast is made to the MTM_Controllers on all hosts requesting a copy. A copy, if it exists, will be sent to that host.

An important factor in the efficiency of the receive operations is message copy distribution. For the resilient class of message object, it is planned initially to have a volatile copy of the message object on each host of the receiver. This concurs with the most traditional view of messages. This will be varied during performance modeling.

6.3.8 Msg_Status Task

The msg_status task determines the status of a particular message by getting the statuses from each of the respective message copies and merging the statuses so that there is one status for each receiver which may be received, not_received, or unavailable.

```
TASK TYPE msg_status IS

  ENTRY msg_status_request_return (return_statuses:  IN MTM_type.msg_opn
                                                         _return_list);

END msg_status_task;
```

This task is the means by which remote statuses are returned to this task.

In the event that not all message copies are available, the msg_task will return to the invoker an "unavailable" status for any receivers whose status

is not known. It is important to note that the msg_status task will not wait forever if one or more copy statuses are not available.

6.3.9 Supporter Task

The supporter task detects the occurrence of a finite set of events, and causes any process that is blocked and waiting on the occurrence of this event to be "awakened". This task performs event detection for senders that are waiting for message copies to be routed to particular hosts, and causes the sender to become unblocked when either the sender times out on that call or when all acknowledgements are received for a message copy being routed to other hosts.

For a receiver, this task detects when a message is available or when his time out has occurred. In the case where a message is available (where available means the message queue of the receiver has a msg_id from a sender), but a copy is not on the host, a copy will be routed to the receiver's host and the receiver will receive it if it is routed before his time out occurs.

There is one supporter task active for a given MTM. It handles event detection for all senders and receivers on that host that specify the wait option on their call with the excep- tion of receivers who specify the wait option but do not wait because a message is available on that host for receipt. Figure 6.11 is a diagram of the tasks that interact with the Supporter task along with the reason for the communication.

```
TASK supporter

  ENTRY gather_receipts_for (sender:  IN kernel.xtnded_uid;
                             msg_id:  IN kernel.xtnded_uid;
                            timeout:  IN POSITIVE;
                           call_uid:  IN kernel.uid);

  ENTRY wait_on_available_message (receiver:  IN kernel.xtnded_uid;
                          receive_from_list:  IN kernel.xtnded_uid_list;
                                    timeout:  IN POSITIVE):

  ENTRY wait_on_msg_copy (receiver:  IN kernel.xtnded_uid;
                            msg_id:  IN kernel.xtnded_uid;
                           timeout:  IN POSITIVE);

  ENTRY wake_up_status (process_id:  IN kernel.xtnded_uid;
                          response:    IN    MTM_type.msg_opn_return);    END
supporter;
```

6.3.10 Waker Task

The supporter task does not directly cause a sender or receiver to be "awakened". When the supporter detects the occurrence of an event to awaken a particular process a waker task is scheduled that returns the result to the Message_Operations task of the sender or receiver which causes the respective sender or receiver to be "awakened" (unblocked) with the results of the original call. This is the end of the sy chronous call for that process.

6.3.11 Remote_Receive_Call and Remote_Receive_Response Tasks

They are part of the interface to the kernel, and as indi- cated by their names, they are responsible for accepting incoming remote calls to the MTM and the incoming responses to remote calls made by the MTM. The word "accepting" in the previous sentence is used loosely because there are no entries into these tasks, rather the tasks invoke procedures in the kernel (get_call and get_resp) that return a call invoked on this MTM or a response to a call made by the MTM, respectively. Figures 6-10 and 6-11 are diagrams of the calls from the Message Type Manager to the kernel routines that interface with the network.

The call or response is unpacked (because it is passed as a bit string from the kernel) and an entry call is made from the respective Remote_Receive_Call (or Response) task to the appropriate task of the MTM, namely the task for which the call or response is intended.

```
TASK remote_receive_call IS

  --This task makes entry calls into various other tasks, but
  --it contains no ACCEPTs.

END remote_receive_call;
```

and,

```
TASK remote_receive_response IS

  --This task makes entry calls into various other tasks, but
  --contains no ACCEPTs.

END remote_receive_response;
```

6.3.12 Process_Message_Queue Task

This task manages a set of message queues for processes that reside on this host. There is one message queue for each process that performs message operations. The message queue (hereafter called pmq) is manipulated through entry calls to insert and delete msg_ids. (These operations are invoked by the send, receive and waker tasks as shown in Figure 6-12).

```
ENTRY delete_msg_id (msg_id:  IN kernel.xtnded_uid;
                   receiver:  IN kernel.xtnded_uid);

ENTRY insert_msg_id (msg_id:  IN kernel.xtnded_uid;
                     sender:  IN kernel.xtnded_uid;
                   receiver:  IN kernel.xtnded_uid);
```

Additionally, there is a "read only" call that returns a msg_id that meets the parameter specifications in the call.

```
ENTRY check_message_available (receiver:  IN kernel.xtnded_uid;
                          receive_from_list:  IN MTM_type.xid_list;
                           which_msg_option:  IN MTM_type.receive_option;
                                     msg_id:  OUT kernel.xtnded_uid;
                                   who_from:  IN kernel.xtnded_uid;
                                  available:  IN BOOLEAN);
```

The receive and waker tasks use this entry to get a msg_id for a message that is available for a receiver. The message indicated by the msg_id will (ultimately) be received by the receiver.

After recovery from a failure, the MTM_Controller notifies this task causing several actions.

```
ENTRY restart_after_failure ( );
```

A database that maintains the pmqs for processes on this host is opened. Once message activity begins, pmqs are moved to memory and accessed. They are moved into memory as they are needed. Maintaining a data base of pmqs is for the integrity and the reliability of the message system. When a pmq in memory is updated, its copy in the data base is updated as well, so that a failure will not render any pmq inconsistent relative to the messages received and the messages waiting to be received.

6.3.13 Message_Object Task

This task performs manipulations of a message object, including creation of the message and accessing parts of the object. These entries are called by the send, receive, waker, and msg_status tasks and the MTM_Controller as is shown in Figure 6-13. As can be shown from Figure 6-13, the Message_Object task manages the volatile message instances on a host, and modifies any incoming message objects to reflect their new residency.

```
ENTRY create (msg_id:  IN kernel.xtnded_uid;
        send_to_list:  IN MTM_type.xid_list;
   reliability_class:  IN MTM_type.rel_classes;
    number_of_copies:  IN POSITIVE;
                text:  IN kernel.message;
    message_instance:  OUT MTM_type.object);
```

The task is passed the information that will compose a message object and assignments are made to the fields of the message object and the object is returned.

```
ENTRY get_msg_status (msg_id:  IN kernel.xtnded_uid;
            number_of_copies:  OUT POSITIVE;
             return_statuses:  OUT MTM_type.msg_opn_return list;
                        done:  OUT BOOLEAN);
```

The message object is retrieved from storage if it is on this host, and the number of copies and the return_statuses are assigned the return_statuses that exist in the message object. Done r turns the boolean value true if the operation completes successfully; otherwise, false is returned.

```
ENTRY get_instance (msg_id:  IN kernel.xtnded_uid;
          message_instance:  OUT MTM_type.object;
                      done:  OUT BOOLEAN);
```

This entry gets a message object from memory through calls to the kernel.

```
ENTRY put_instance (msg_id:  IN kernel.xtnded_uid;
          message_instance:  IN kernel.xtnded_uid;
                    medium:  IN MTM_type.storage
                      done:  OUT BOOLEAN);
```

This entry causes a message object to be put into memory through calls to the kernel.

```
ENTRY update_receipt (msg_id:  IN kernel.xtnded_uid;
                    receiver:  IN kernel.xtnded_uid);
```

This causes a message object with msg_id to be updated to reflect that the indicated receiver has received the message. If the message copy is local, a change is made locally to reflect the status of the receiver. If the message copy is remote, a call is broadcast to remote MTM_controllers (to the msg_received entry) which causes any existing copies to be updated.

This concludes the discussion of the user interface to the MTM, the tasks of the MTM and the means by which incoming remote calls and responses are received from the kernel. One point of importance, with regard to the operations, is that the operations of the send, receive, and msg_status processes are performed as transactions. Because of this, message operations have well- defined properties and the consistency of message objects is maintained. To the user, the operations will be perceived as having a particular effect if they succeed and no effect if they do not succeed.

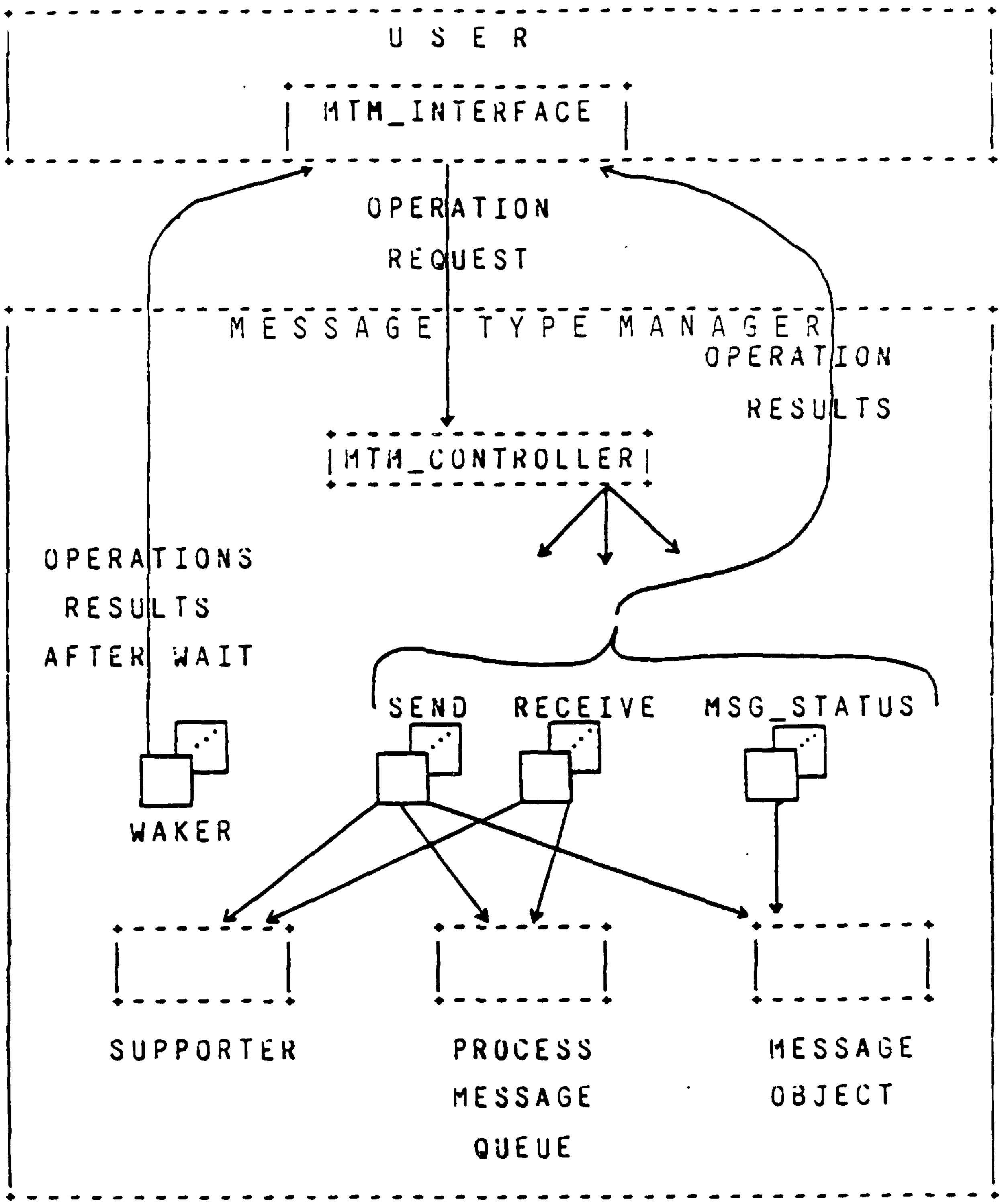
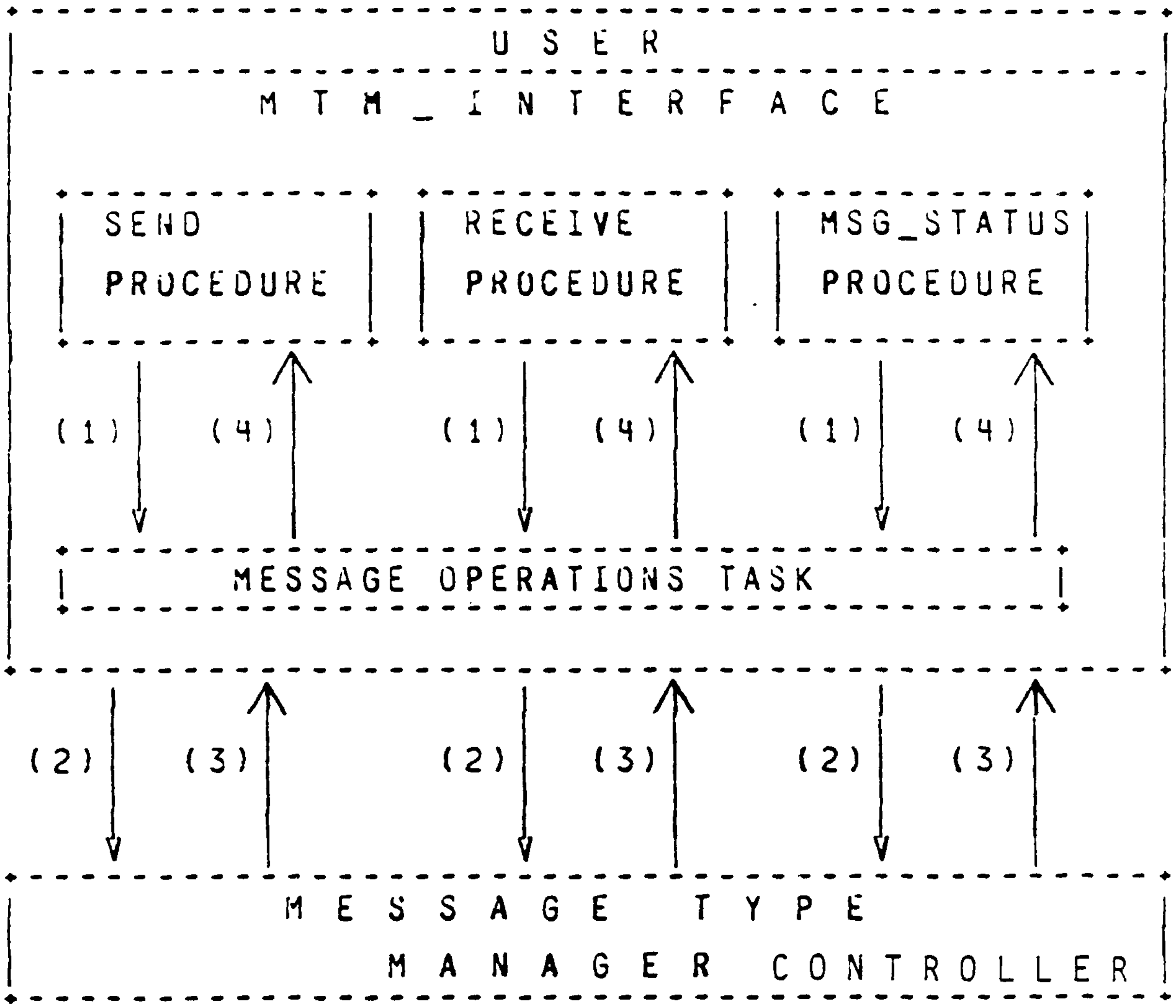


Figure 6-1 Components of Message Management

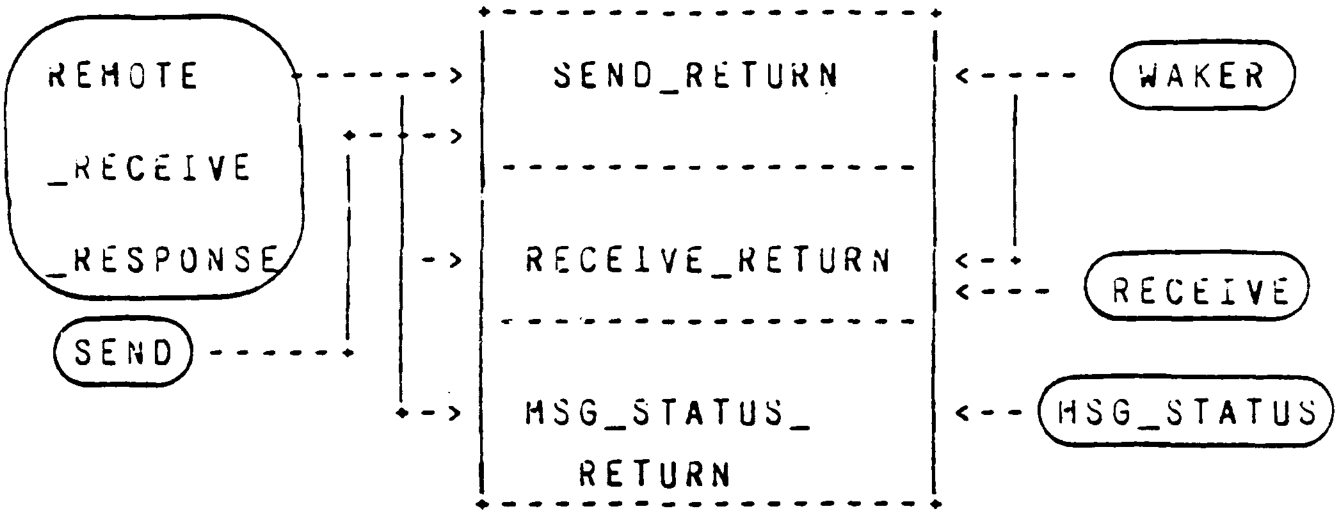


(1) REQUEST OPERATION

(2) ROUTE REQUEST

(3), (4) RETURN RESULT

Figure 6-2 The Sequence of Events Between the User's MTM_Interface and the MTM_Controller



<u>ENTRIES OF THE</u>

. <u>MESSAGE OPERATIONS TASK</u>

. CONTAINED WITHIN THE MTM_INTERFACE

TO THE USER

Figure 6-3

Entry Call/Task Relationships for the Message_Operations Task

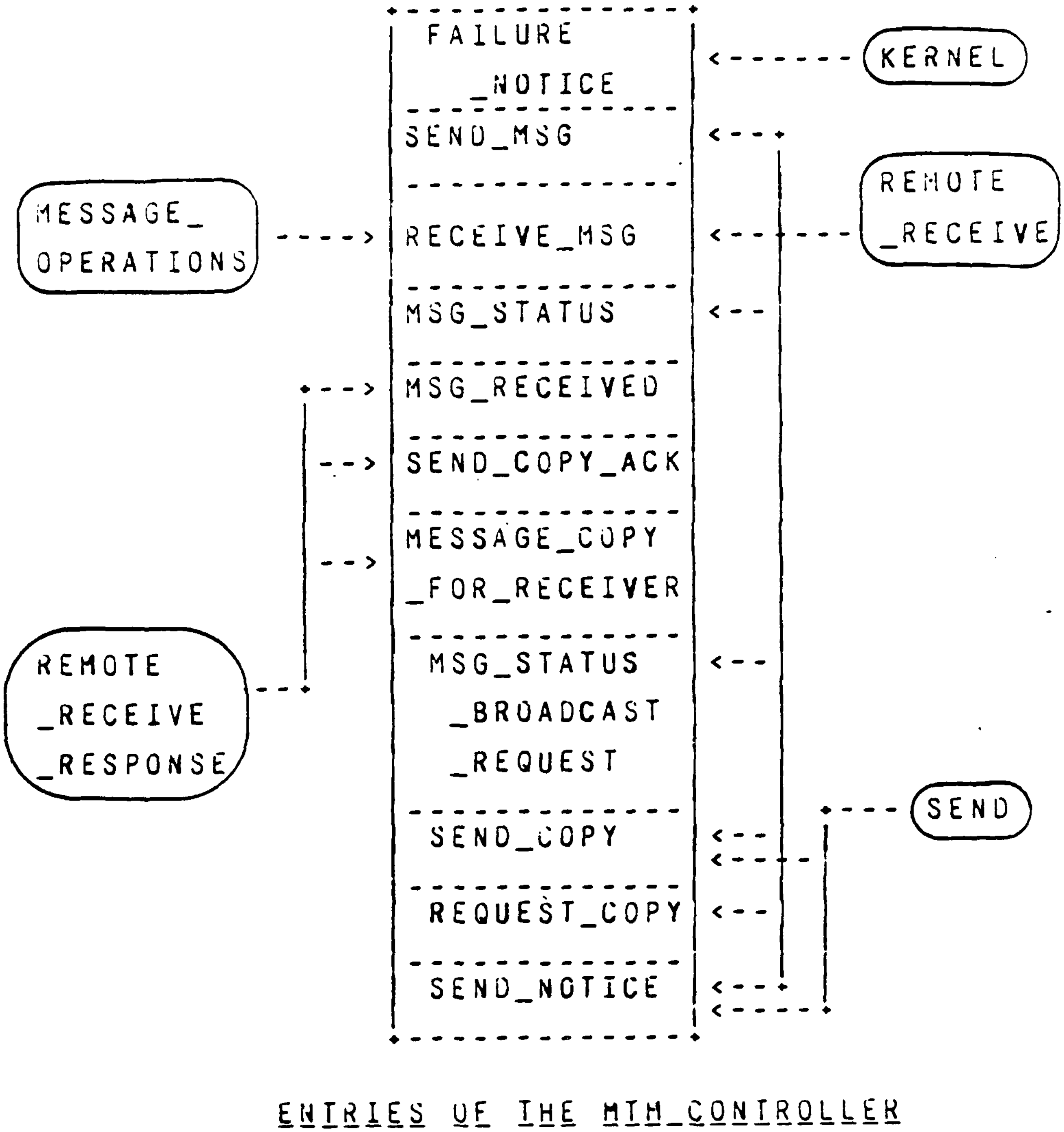
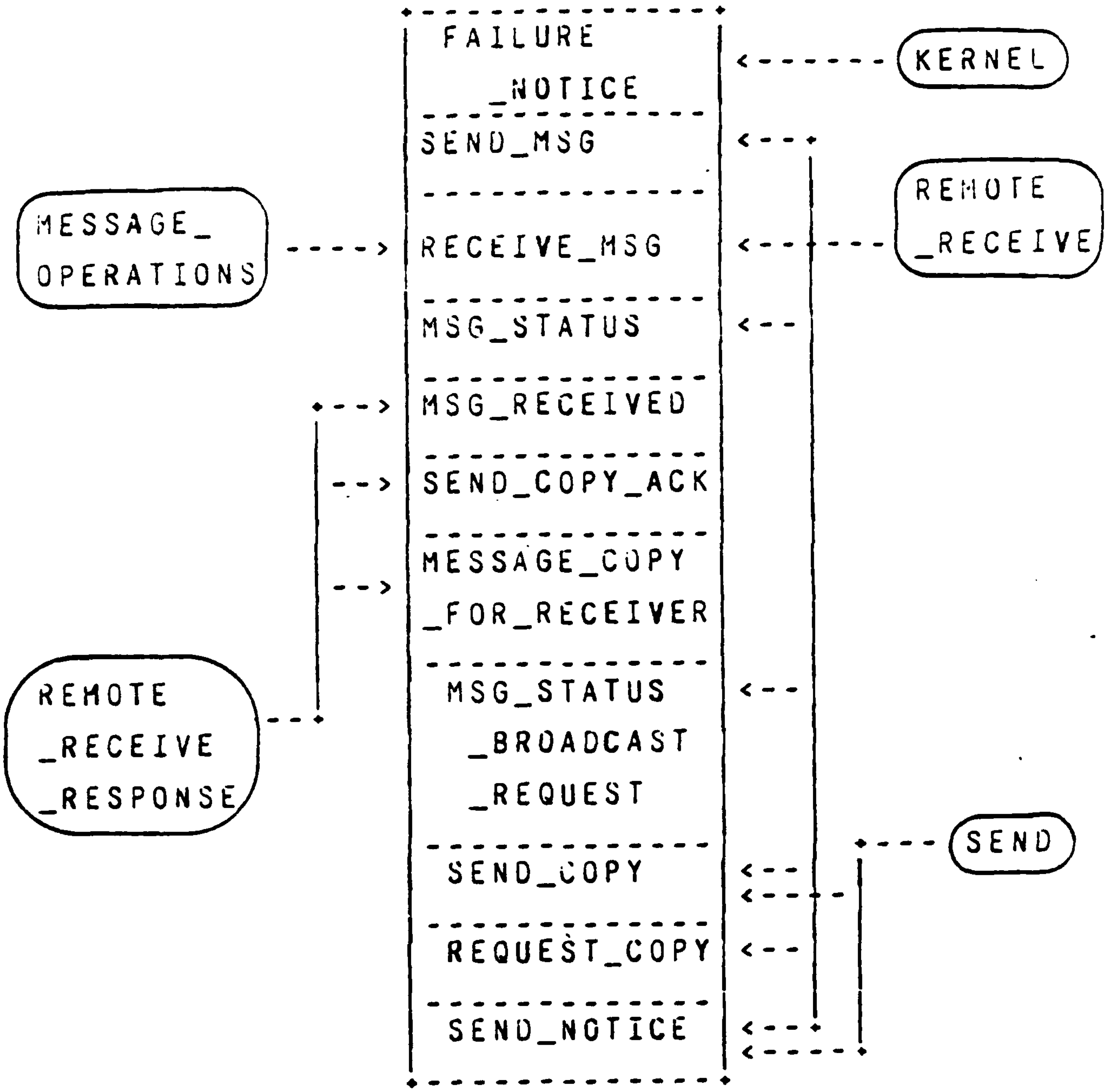


Figure 6-5 Entry Call/Task Relationships for the MTM_Controller

ENTRY CLASS	ENTRIES	TASKS SCHEDULED
MESSAGE OPERATIONS	SEND_MSG RECEIVE_MSG MSG_STATUS	SEND RECEIVE MSG_STATUS
ACKS	MSG_RECEIVED SEND_COPY_ACK	MESSAGE_OBJECT WAIT_QUEUE
ROUTING COPIES	MESSAGE_COPY_FOR_RECEIVER SEND_COPY REQUEST_COPY	MESSAGE_OBJECT ----- -----
MESSAGE QUEUE	SEND_NOTICE FAILURE_NOTICE	PROCESS_MESSAGE_QUEUE

Figure 6-4 MTM_Controller Entries and the Tasks Effected by the Entry



ENTRIES OF THE MTM_CONTROLLER

Figure 6-5 Entry Call/Task Relationships for the MTM_Controller

MESSAGE OBJECT OPERATIONS	KERNEL INTERFACE	TASK SCHEDULING
MSG_STATUS	REMOTE_RECEIVE_CALL	MTM_CONTROLLER
RECEIVE	REMOTE_RECEIVE_RESPONSE	
SEND		
SUPPORTER WAKER		
MESSAGE_OBJECT		
PROCESS_MESSAGE_QUEUE		

Figure 6-6 Tasks of the Message Type Manager

```
              MTM_CONTROLLER  MTM_CONTROLLER
                     OF              OF
SENDER            SENDER          RECEIVER       RECEIVER

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  |                |                  |               |
  |          SEND  | MSG              |               |
  | - - - - - - -> |                  |               |
  |                |                  |               |
  |                |            SEND  | COPY          |
  |                | - - - - - - - -> |               |
  |                |                  |               |
  |     SEND COPY  | ACK              |               |
  |                | <- - - - - - - - |               |
  |                |                  |               |
  |                |         RECEIVE  | MSG           |
  |                |                  | <- - - - - - -|
  |                |                  |               |
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 6-7 Directed Receive

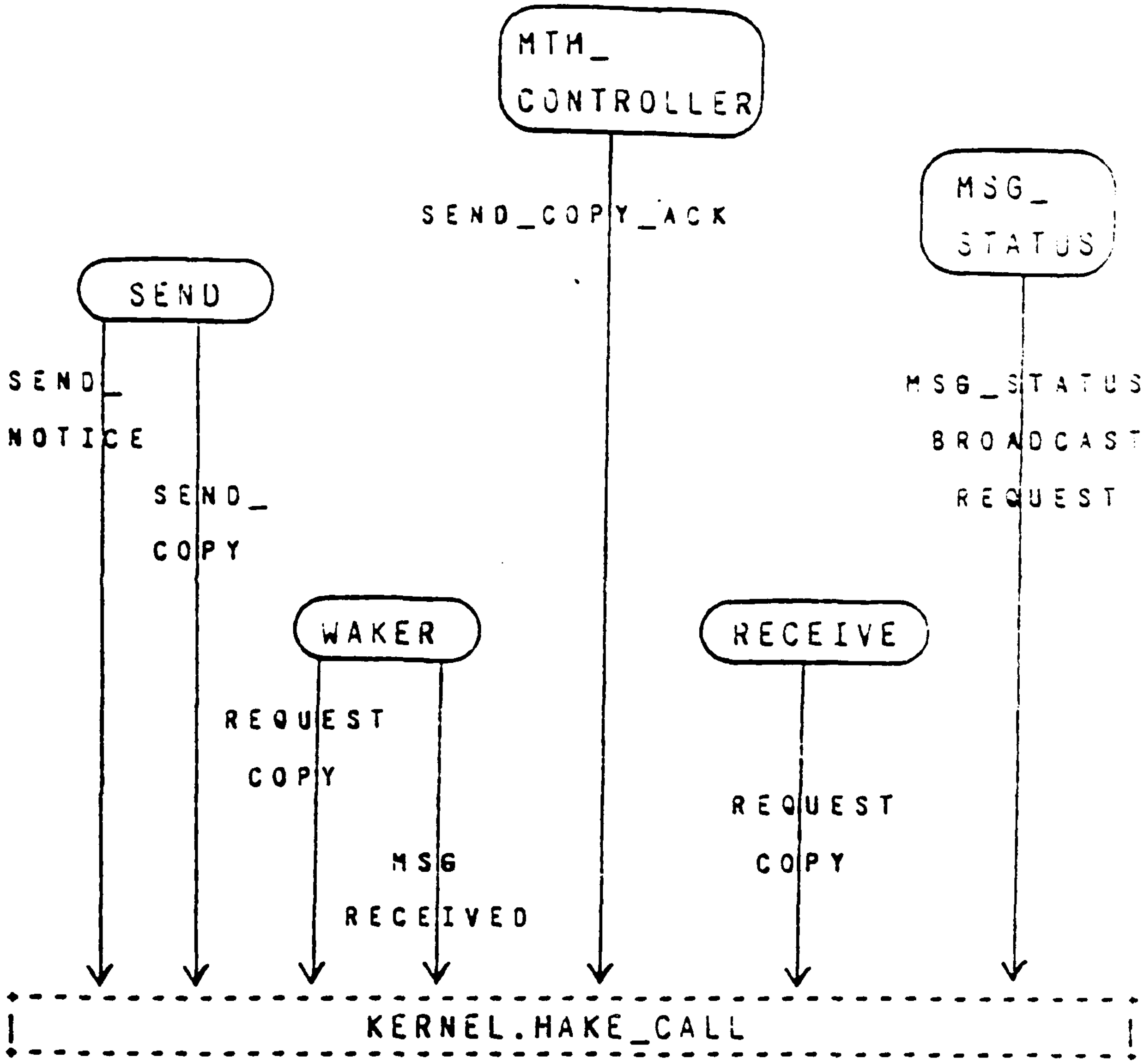
```
            MTH_CONTROLLER  MTH_CONTROLLER
                  OF              OF
SENDER         SENDER          RECEIVER        RECEIVER

-----------------------------------------------------------
   |            |                   |                 |
   |        SEND|MSG                |                 |
   | ---------> |                   |                 |
   |            |                   |                 |
   |            |               SEND|NOTICE           |
   |            | ----------------> |                 |
   |            |                   |                 |
   |            |            RECEIVE|MSG              |
   |            |                   | <-------------- |
   |            |                   |                 |
   |     REQUEST|COPY               |                 |
   |            | <---------------- |                 |
   |            |       MSG COPY FOR|RECEIVER         |
   |            | ----------------> |                 |
   |            |                   |                 |
   |         MSG|RECEIVED           |                 |
   |            | <---------------- |                 |
   |            |                   |                 |
-----------------------------------------------------------
```

Figure 6-8 Receive on Demand

	ENTRIES OF THE SUPPORTER TASK	OPERATION STATE
SEND	GATHER_RECEIPTS_FOR	SEND WITH WAIT OPTION
MTH_CONTROLLER	MARK_SENDER_COPY_COUNT	FOR A WAITING SENDER
RECEIVE	WAIT_ON_AVAILABLE_MESSAGE	RECEIVE WITH WAIT OPTION
RECEIVE	WAIT_ON_MESSAGE_COPY	DITTO, AND MESSAGE COPY ON ANOTHER HOST
WAKER	WAKE_UP_STATUS	DENOTES WHETHER OR NOT THE PROCESS IS AWAKENED

- PERFORMS DETECTION OF EVENTS
- CAUSES WAKER TO UN_BLOCK WAITING USER PROCESSES

Figure 6-9 Entry Call/Task Relationships for the Supporter Task



- CALLS TO REMOTE HOSTS ARE MADE VIA KERNEL.MAKE_CALL
- THERE MAY OR MAY NOT BE A RESPONSE TO A CALL

Figure 6-10 Calls From the Message Type Manager to Kernel.Make_Call

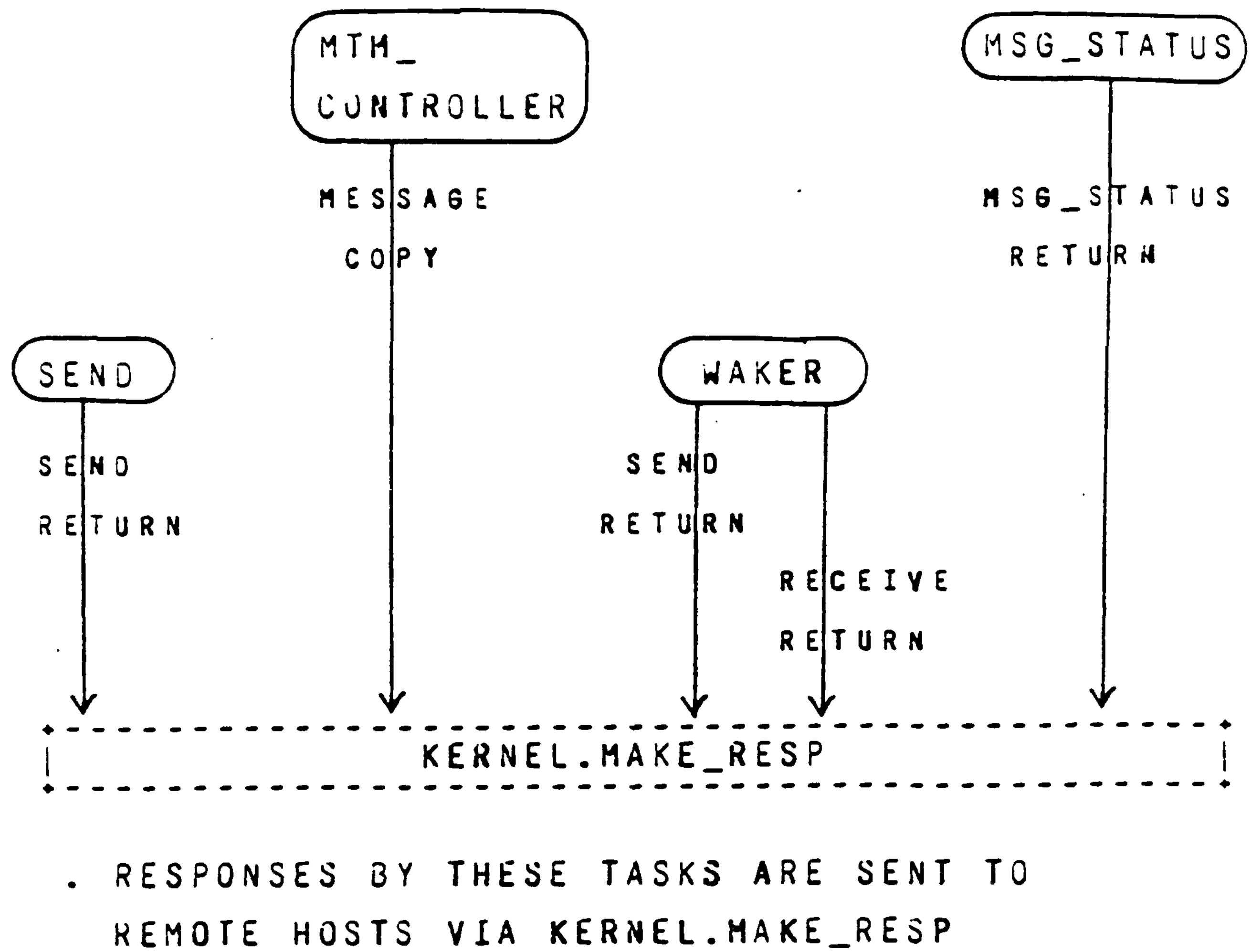
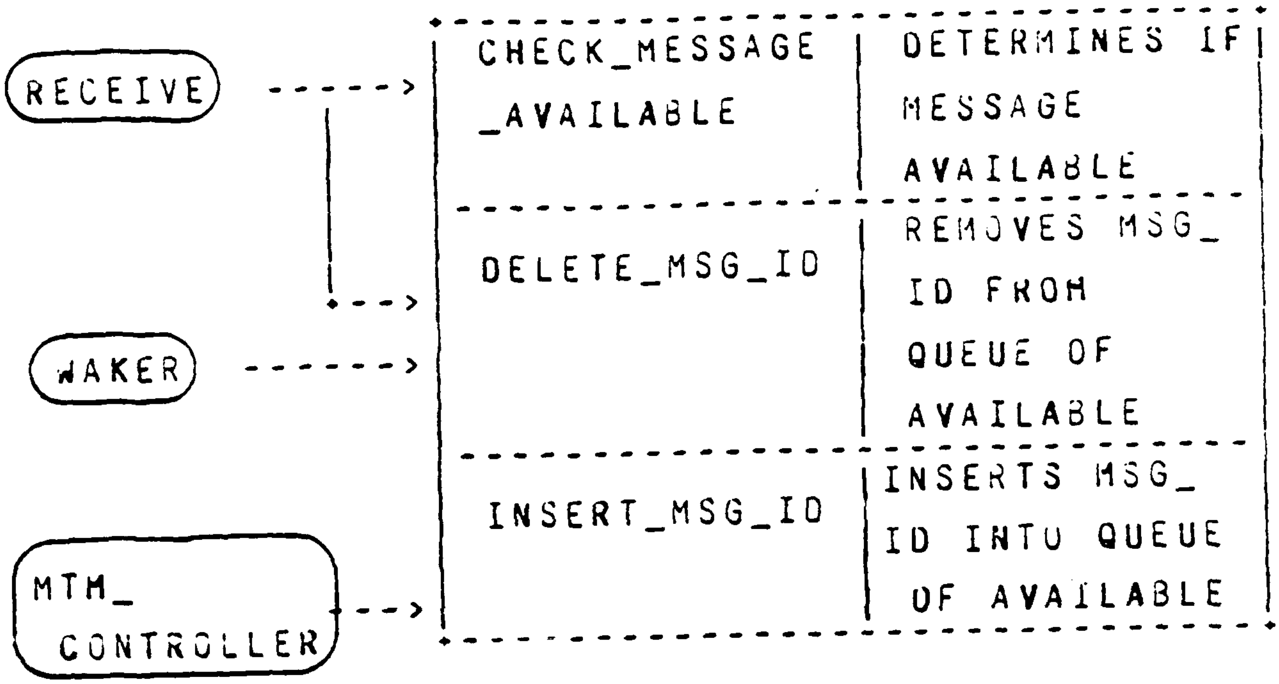


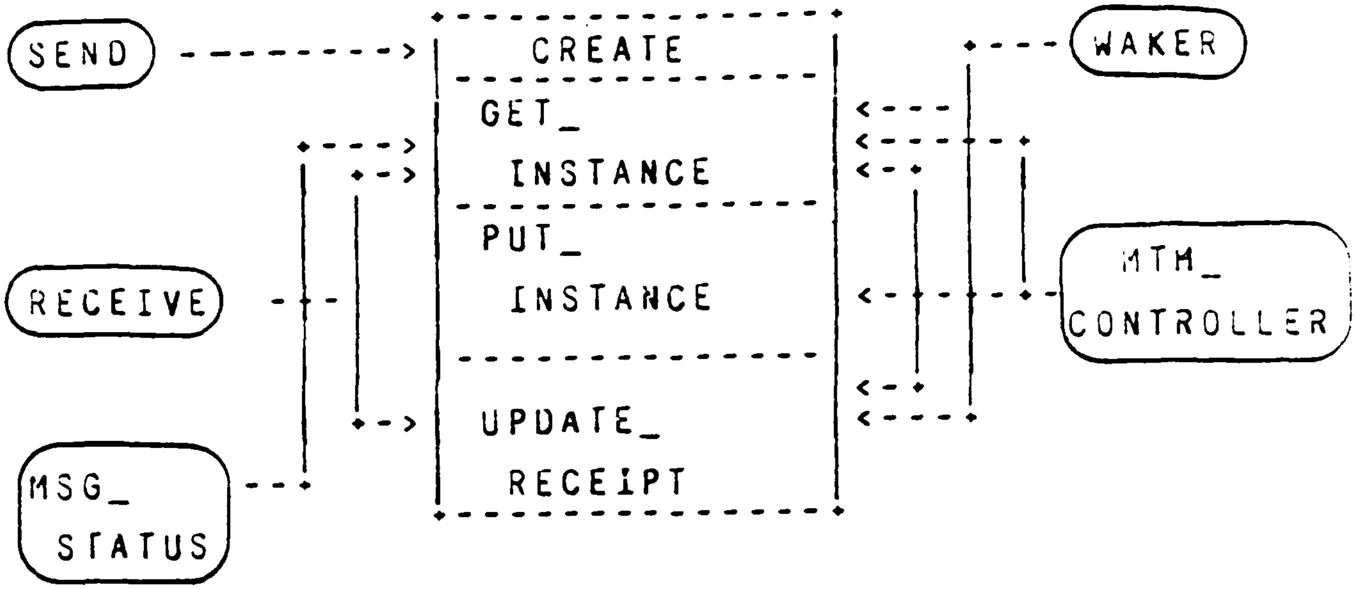
Figure 6-11 Calls From the Message Type Manager to Kernel.Make_Resp



ENTRIES OF THE
PROCESS_MESSAGE_QUEUE TASK

- MAINTAINS MSG_ID QUEUES, ONE FOR EACH PROCESS ON THE HOST
- THE MSG_ID IS IN A QUEUE WHEN THAT MESSAGE IS AVAILABLE FOR A PROCESS

Figure 6-12 Entry Call/Task Relationships for the Process_Message_Queue Task



ENTRIES FOR THE
MESSAGE OBJECT TASK

- MANAGES INSTANCES OF VOLATILE MESSAGES ON A HOST
- MODIFIES A MESSAGE OBJECT WHEN IT IS RECEIVED

Figure 6-13 Entry Call/Task Relationships for the Message_Object Task

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management, information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic, maintainability, and compatibility.